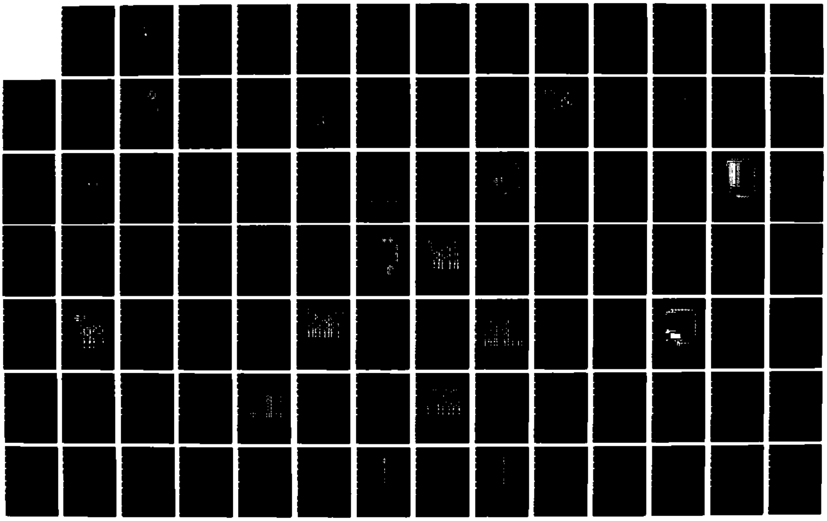
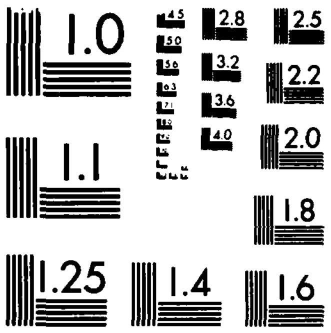
AD-A161 098 VLSI (VERY LARGE SCALE INTEGRATED CIRCUITS) DESIGN WITH THE MACPITTS SILICON COMPILER(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R C LARRABEE SEP 85 1/3

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

VLSI DESIGN WITH THE MACPITTS
SILICON COMPILER

by

Robert C. Larrabee

September 1985

Thesis Advisor: D. E. Kirk

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A161098	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* VLSI Design With The MacPitts Silicon Compiler		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1985
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Robert C. Larrabee		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943-5100		12. REPORT DATE September 1985
		13. NUMBER OF PAGES 265
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

VLSI, Silicon Compiler, MacPitts, VLSI Design

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

An analysis of the MacPitts silicon compiler is presented. The emphasis of the analysis is on the interrelationship between algorithmic syntax and resulting circuit structure. Errors inherent to the silicon compiler are investigated, and corrections to the errors are proposed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

VLSI Design With The MacPitts Silicon Compiler

by

Robert C. Larrabee
Lieutenant, United States Navy
B.S., University of Texas at Austin, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 1985

Author: [illegible]
Robert C. Larrabee

Approved by: [illegible]
D. E. Kirk, Thesis Advisor

[illegible]
H. H. Loomis, Second Reader

H. B. Rigas, Department of
Electrical and Computer Engineering

[illegible]
John N. Dyer,
Dean of Science and Engineering

ABSTRACT

An analysis of the MacPitts silicon compiler is presented. The emphasis of the analysis is on the interrelationship between algorithmic syntax and resulting circuit structure. Errors inherent to the silicon compiler are investigated, and corrections to the errors are proposed.

TABLE OF CONTENTS

I. INTRODUCTION

The purpose of silicon compilation is to allow faster design of integrated circuits. Silicon compilation frees the designer from the basic layout, routing, and circuitry concerns inherent to integrated circuit design. The MacPitts silicon compiler does this by designing an integrated circuit chip from a behavioral specification input.

Previous work at the Naval Postgraduate School investigated applications of the MacPitts silicon compiler to design of pipelined digital adders [Ref. 1] and multipliers [Ref. 2]. Work by Froede [Ref. 3] showed the limitations of MacPitts, in its inability to produce fast VLSI chips. This deficiency is due primarily to the layout scheme (circuit structure) which MacPitts uses.

This thesis investigates the interrelationship between MacPitts algorithmic syntax and resulting circuit structure. MacPitts partitions the chip functionally as shown in Figure 1.1. The data path is at the top, and performs numerical operations and combinational logic tests. The control path is at the bottom, and performs decisions which direct data path operations.

Chapter II considers combinational logic in both the data path and control path. The effects of syntax on combinational logic structures are investigated

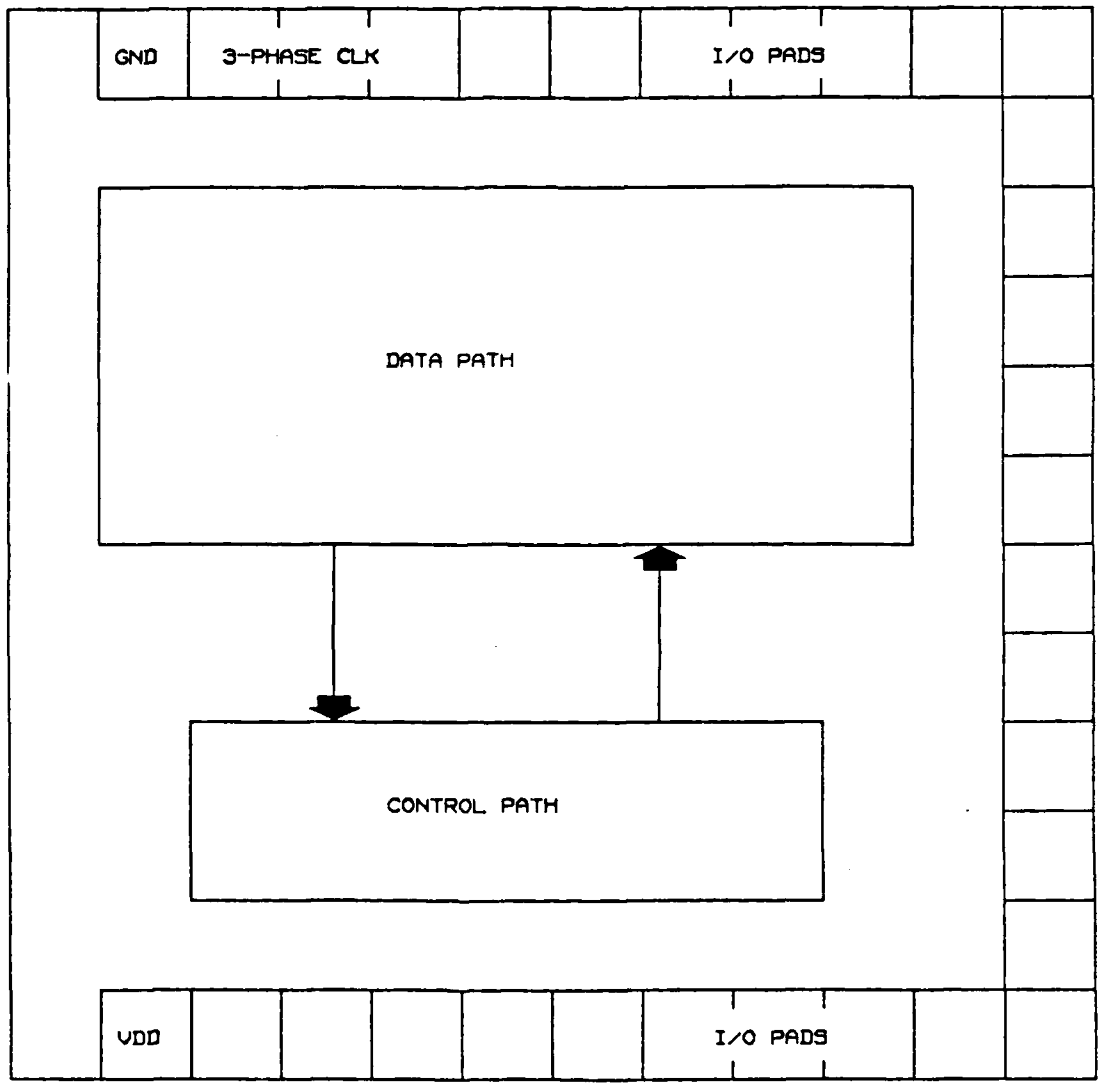


Figure 1.1 MacPitts Chip Functional Block Diagram

qualitatively, and inefficiencies and limitations of implementation are noted. The basic data path organelles (fundamental combinational logic structures) are also investigated.

Chapter III is a quantitative treatment of functionally equivalent circuits in the data path and control path. A five-input AND gate is created in both the data path and thecontrol path, and a comparative analysis is performed. The results are extended to similar data path combinational logic structures.

Chapter IV investigates MacPitts sequential logic. A Gray code-to-binary serial decoder is designed, and a functional analysis is performed. The relationship between syntax and circuit structure is emphasized, with an alternate solution considered. A blackjack game chip is presented as a more elaborate MacPitts finite state machine (FSM), and its structure is contrasted to that of the Gray code decoder. The Mead-Conway highway-farmroad traffic light controller [Ref. 4:p.81] problem is solved with a MacPitts design, and an alternate solution is offered.

Chapter V is a quantitative comparison of a MacPitts design with a handcrafted equivalent. The Mead-Conway traffic light controller design from Chapter IV is compared to a computer-aided engineering (CAE)-designed variant, which has a programmed logic array (PLA) FSM. The designs are compared for speed, size, and power comsumption.

Chapter VI is a design example. A design cycle for MacPitts is developed, and illustrated with the Hamming 15/4 error detector/corrector [Ref. 5]. The prototype (first model) and archetype (chief model) algorithms and chip layouts are provided. An analysis of the alternate designs is given, and a basis for choosing the archetype is proposed. The Hamming 15/4 error detector/corrector is then designed based on the archetype, and analyzed with available CAD tools.

Chapter VII is a summary of errors detected in the MacPitts silicon compiler and suggestions for enhancement. The errors and suggestions are cross-referenced to MacPitts source code where possible.

II. COMBINATIONAL LOGIC STRUCTURES IN THE MACPITTS SILICON COMPILER

Inasmuch as the MacPitts algorithm creates combinational logic functions, it would be helpful to know how it does this. Does there exist an explicit directive to the LISP object file which calls and implements the logical functions requested, or are they implicitly specified? If the latter is true, it would suggest simpler source algorithms could be written to specify the circuit function. If the former case is true, then more lengthy algorithms are required, but the circuit designer has more latitude for direct control and optimization of layout.

A. COMBINATIONAL LOGIC CIRCUITS IN THE DATA PATH

Combinational logic structure instantiation in the data path of a MacPitts generated chip is directed by the data-path.lisp file in the MacPitts source code. The data-path.lisp file calls specific functional units called organelles from the organelles.lisp file to implement the desired logic. These LISP files are compiled under the Liszt compiler and linked to the rest of the compiled MacPitts files by the available Makefile routine. The resulting 1.6 Megabyte binary image constitutes the integrated MacPitts silicon compiler.

1. <u>The Basic Chip Frame</u>

The initial investigation consisted of the MacPitts-generated design frame called wire.mac. The algorithm to create this structure is shown in Figure 2.1.

```
;WIRE.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF NO
;FUNCTION BY MACPITTS SILICON COMPILER
(program' wire 1
        (def 1 ground)
        (def ain port input (2))

        (def res port output (3))
        (def 4 phia)
        (def 5 phib)
        (def 6 phic)

        (def 7 power)
        (always
                (setq res    ain))))
```

Figure 2.1 Wire.mac

The extension .mac refers to a MacPitts algorithm. MacPitts is taken to refer to the silicon compiler, the psuedo-LISP language which it uses, and the LISP source routines which constitute the silicon compiler. To avoid confusion, the MacPitts driver routines written by the chip designer will be referred to as algorithms. Other meanings of the term MacPitts will be clarified by context.

MacPitts produces a seven pad chip, routing the input directly to the output without clocking. The three phase clocking is not required for this circuit, so the clock runs all terminate within the chip frame without connections as shown in Figure 2.2. The three phase clock must be specified in the algorithm, however, and the clock traces are produced whether they are used or not. Note that the pads are placed around only three sides of the chip,

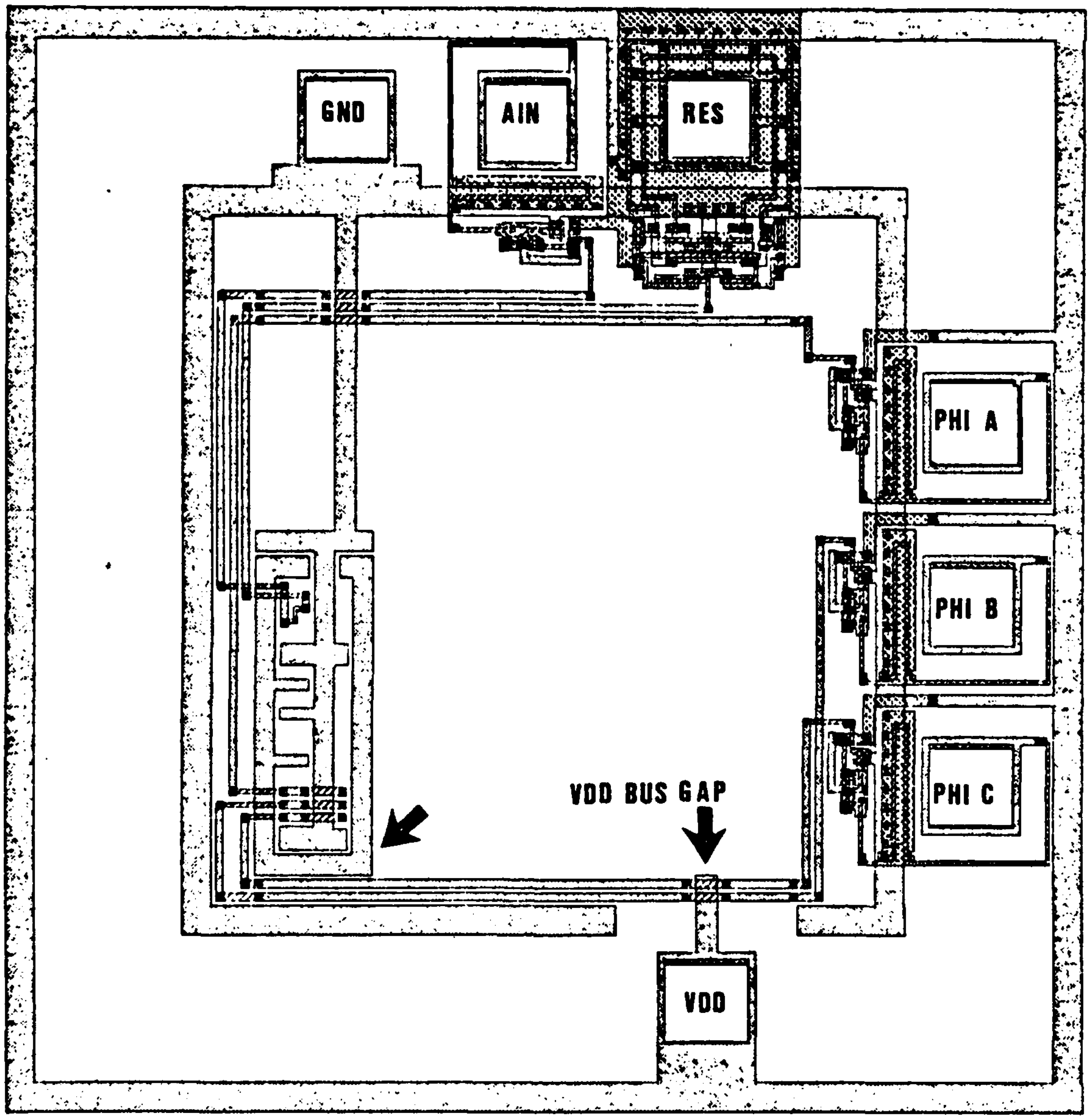


Figure 2.2 Wire.cif

and the clock pads are also placed in the order specified in the driver algorithm (Figure 2.1). Furthermore, neither the clock traces nor the signal lines takes a direct route to its destination. Even though these lines are all metal, the excess lengths induce a lessening of maximum chip speed due to capacitance. This topic will be treated in a later chapter. The data path Vdd-ground comb does not connect with the Vdd rail at bottom left on the stipple plot. This is common with very small data path chips, and the error can be corrected in Caesar or a similar VLSI graphics editor.

2. A Data Path Inverter

The next program, macnot.mac shown in Figure 2.3, specified a logical NOT function. As expected, MacPitts used a single inverter of 4:1 ratio in the data path. The input which is on the top left diffusion line in Figure 2.4 runs to the gate of the NMOS inverter via a metal and diffusion routing, and the inverted output comes out on a polysilicon line from the far right of the circuit. It was also noted that the logical integer specification is required for NOT, i.e. , one must use [word-not] rather than [not]. The reason for this is given in Southard [Ref. 6:pp. 47-48], which indicates that integer logical operators must be used on word elements, (ports and registers), and Boolean logical operators on control elements (flags and signals). The logical Boolean specification [not] is used on flags, input signals, and internal signals but it is not used for input ports or register contents. In either Boolean or integer data types, the NOT function takes a single value, as would be expected.

The syntax of the driver algorithm (the .mac file) is data-type sensitive, in a similar manner as Fortran is sensitive to the integer and floating point data types. The two data types (from the programming perspective) are Boolean and integer. Each data type is treated differently by the MacPitts compiler, and each requires a different syntax for the equivalent function. An example will clarify this distinction:

FUNCTION	DATA TYPE	ALGORITHMIC STATEMENT
NOT	Boolean	(not a)
NOT	integer	(word-not a)
AND	Boolean	(and a b)
AND	integer	(word-and a b)

The fundamental difference in data types is argument length. Boolean data are of single bit length, whereas integer data are of word length (one bit or greater). Integer type data operations all occur in the data path of a MacPitts design, and Boolean operations all occur in the control path.

In Figure 2.3, the data type is declared in the DEF statement, the form of which is

```
(def <name> <function> <input, output, or internal>
<pin number(s)>)
```

```
;MACNOT.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<not>  FUNCTION BY MACPITTS SILICON COMPILER
(program macnot 1
         (def 1 ground)
         (def a port  input (2))
                      ;ain=input//res=output
         (def b port  output (3))
         (def 4 phia)
         (def 5 phib)
         (def 6 phic)
                      ;must show 3-phs clk,even if not used
         (def 7 power)
         (always
                           (setq b  (word-not a))))
```

Figure 2.3 Macnot.mac

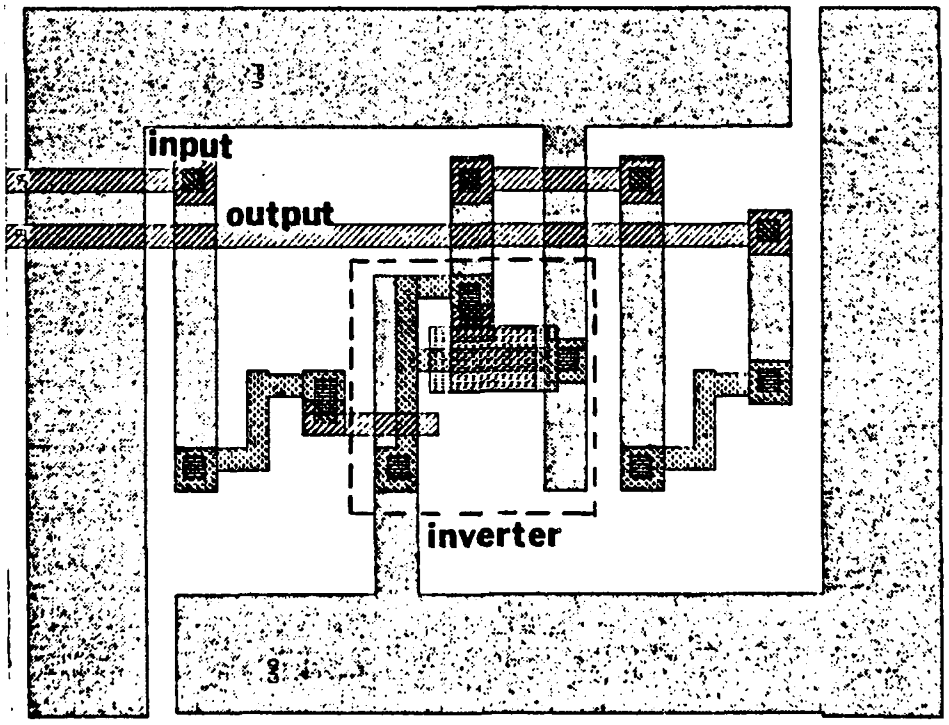


Figure 2.4 Data Path Inverter

where the name is any ASCII character string, the function can be either port, signal, register, or flag. The next field determines where the data is applied, and for most circuits is either input or output. The pin number is required for all input and output data. The data type is determined by the function field. Signals and flags are Boolean data, ports and registers are integer (word length) data. The subsequent MacPitts forms in the driver algorithm must agree in type with the DEF declarations.

If an incorrect data type specification is used, MacPitts generates an appropriate error diagnostic at compilation time. For instance, if one were to define the inputs hot and cold as Boolean type and attempt integer operations on them as follows

```
(def hot signal input 5)
(def cold signal input 6)
              .
              .
(setq warm (word-nor hot cold))
              .
              .
              .
```

the following diagnostic would result at compilation time:

```
Error:logical   coercion   to   integer   not   implemented
yet
```

Similarly, if Boolean operations are attempted on integer data, the following diagnostic results at compilation time:

```
Error:Boolean conversion not implemented yet
```

MacPitts error diagnostics can be quite confusing to the inexperienced user. It is suggested that one peruse the lincoln.lisp, h1.grep, and compmesg.lisp files of the MacPitts source code to gain insight into the cause of specific diagnostic messages.This can be easily done on-line under the BSD Unix operating system. The grep feature (pattern search and recognition) is used. The general command format is

```
grep <search pattern> <file to search>.
```

For example, if one attempted Boolean operations on a register (an integer-valued data type) in MacPitts, the second diagnostic given above would result. To locate the source of this message, change directory to the residence of MacPitts source code and issue the Unix command

```
grep boolean *.*
```

to locate all occurrences of the word boolean. Caution is advised in issuing the grep command. If a very common word is searched for, the search may take quite a long while, and the results may not be very helpful. The search capability of the grep command is limited though, as explained in the BSD Unix manual.

3. A Data Path OR Gate

Next a MacPitts routine was written to generate a two input OR gate in the data path. Again, the integer data specification is required (see Figure 2.5).

```
;MACOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<or>  FUNCTION BY MACPITTS SILICON COMPILER//2 input gate//
(program macor 1
        (def 1 ground)
        (def a port  input (2  ))
                     ;a,b=inputs//c=output
        (def b port input (3))
        (def c port  output (4 ))
        (def 5 phia)
        (def 6 phib)
        (def 7 phic)
        (def 8 power)
        (always
        (setq c ( word-or  a   b  )  )  )  )
```

Figure 2.5 Macor.mac

The resulting circuit extracted from the chip is depicted in Figure 2.6. The OR function is implemented as a NOR gate followed by an inverter. Figure 2.7 shows the gate equivalent of a two input data path OR structure. The two inputs to the NOR gate come in on the left top of the circuit, the output is then inverted to yield a logical OR function, and the output of the inverter is routed from the left back out on the poly line below and parallel to the input tracks. This routing scheme (river routing) is determined by the MacPitts source code, and the chip designer has no control over it. All chip inputs and outputs are routed inside the main ground bus, with little regard to minimizing trace length (see Figure 2.2). So an OR gate in the data path of MacPitts is constructed from a two input NOR gate with an inverter on the output, and the inputs and outputs all connect the data path from the left side.

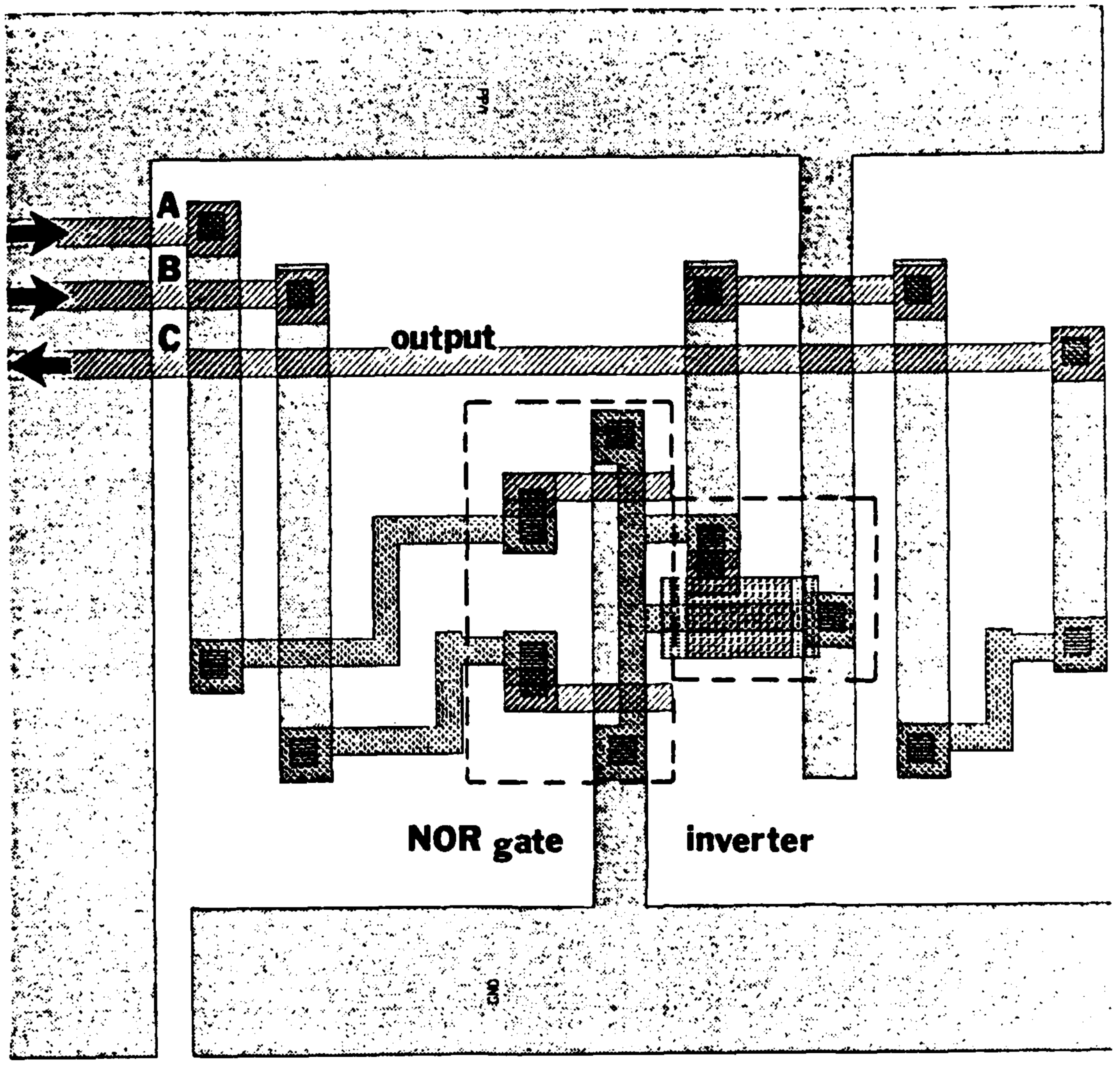


Figure 2.6 Data Path OR Gate

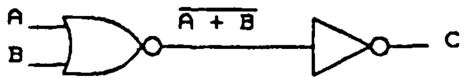


Figure 2.7 Gate Equivalent of Figure 2.6

4. A Data Path NOR Gate

A two input data path NOR function is shown in Figure 2.8. The resulting circuit in Figure 2.9 shows instantiation as a two input 8:1 NOR gate, with the inputs A, B, at top left and the result, C, at bottom left. If two inputs are permissible, are more? Does MacPitts know to adjust the transistor k values for multiple input gates? A two input NOR chip was specified in the algorithm, and MacPitts created a two input NOR gate. So explicit circuit specification has been realized so far in the MacPitts chip data path. When the algorithm specifies a NOR function, a NOR gate is instantiated. As will be discussed later, this is not the case in the control path.

```
;MACNOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<nor>  FUNCTION BY MACPITTS SILICON COMPILER//2 input gate//
(program macnor 1
        (def 1 ground)
        (def a port  input (2  ))
                     ;a,b=inputs//c=output
        (def b port input (3))
        (def c port  output (4 ))
        (def 5 phia)
        (def 6 phib)
        (def 7 phic)

                     ;must show 3-phs clk,even if not used
        (def 8 power)
        (always
        (setq c ( word-nor  a   b  )  )  )  )      )
```

Figure 2.8 Macnor.mac

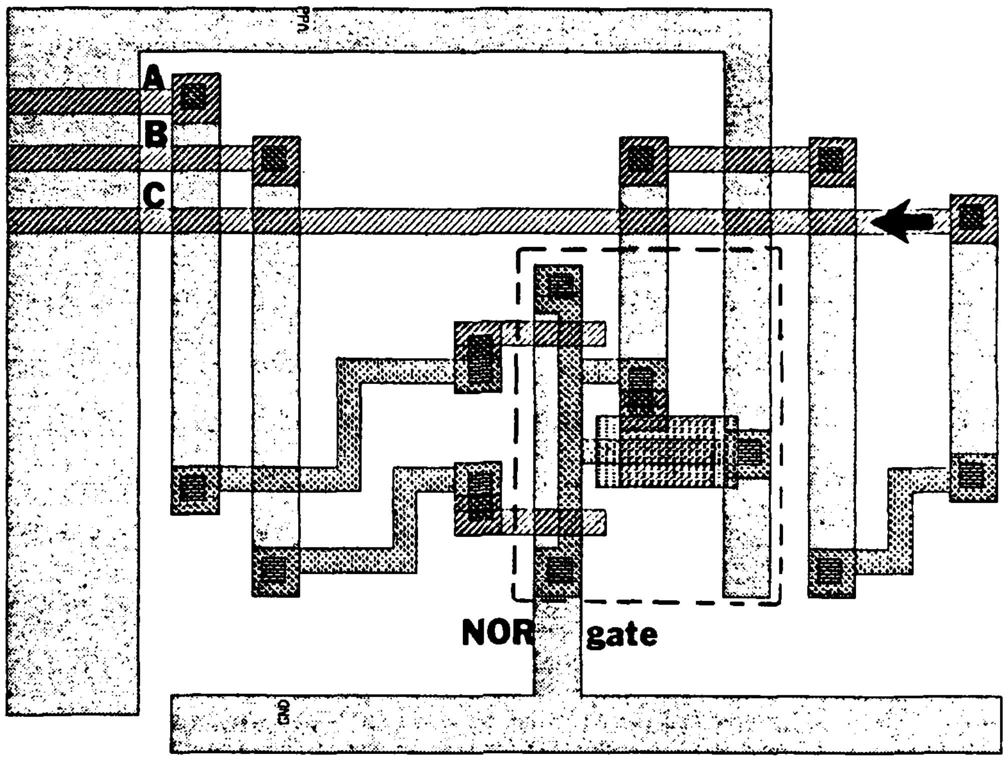


Figure 2.9 Data Path NOR Gate

5. A Four Input NOR Structure In The Data Path

Figure 2.10 shows the MacPitts algorithm to generate a four input NOR structure (not the functional equivalent of a four input NOR gate) in the data path. The MacPitts form used was

```
  (setq out(word-nor a(word-nor b(word-nor c d)))
```

where setq is the LISP assignment operator, out is the output port, a,b,c,and are the inputs, and all data is of integer (word) type. The prefix-operator nature of LISP syntax [Ref. 6:p. 47] indicates the logical operation which this gate will perform. Figure 2.11 shows the layout of the circuit MacPitts produces from this algorithm, and Figure 2.12 depicts the gate-level equivalent.

Note the topology. two inputs to the first NOR gate, its output and another input to the next NOR gate and repetition to the third level. The output comes from the last (rightmost) NOR gate.

This structure will not be the functional equivalent of a four input NOR gate. As the LISP-like syntax suggests, the NOR of four inputs is not equivalent to the cascading of two input NORs.

```
;FOURNOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<nor>  STRUCTURE BY MACPITTS SILICON COMPILER//4 inputs//
(program fivnor 1
        (def 1 ground)
        (def a port input (2))
        (def b port input (3))
        (def c port input (4))
        (def d port input (5))
        (def e port input (6))
        (def outr port output (7))
        (def 8 phia)
        (def 9 phib)
        (def 1Ø phic)
        (def 11 power)
        (always
           (setq outr
             (word-nor a(word-nor b(word-nor c d))))))
```

Figure 2.10 Fournor.mac

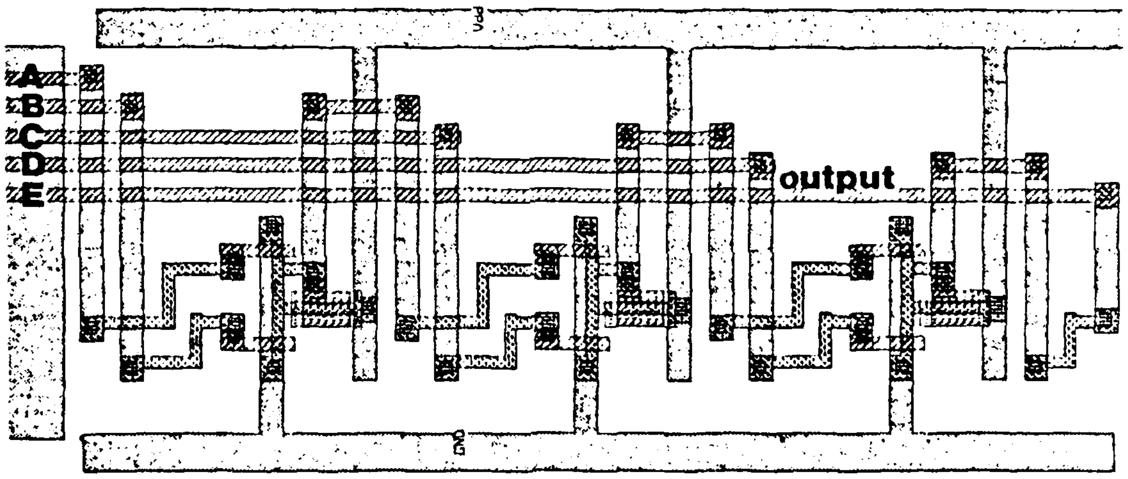


Figure 2.11 Data Path Fournor Circuit

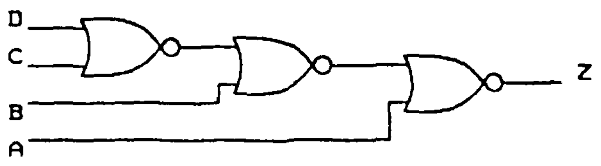


Figure 2.12 Gate Equivalent of Fournor Circuitry

6. A Data Path AND Gate

These observations raise the question of how a two input data path AND gate would be constructed by MacPitts. The (word-and x y) integer expression is required to implement this circuit algorithmically, and a reasonably compact circuit is expected. Figure 2.13 shows the MacPitts algorithm to create the two input bit AND function in the data path.

```
;MACAND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<AND>  FUNCTION BY MACPITTS SILICON COMPILER//2 input gate//
(program macand 1
        (def 1 ground)
        (def a port  input (2  ))
        (def b port input (3))
        (def c port  output (4 ))
        (def 5 phia)
        (def 6 phib)
        (def 7 phic)
        (def 8 power)
        (always
        (setq c ( word-and  a   b )  )  )  )
```

Figure 2.13 Macand.mac

The AND chip is implemented as a two input 4:1 NAND gate, the output of which drives a 4:1 inverter. The stipple plot of this circuit is shown in Figure 2.14, and its gate level equivalent is shown in Figure 2.15. In Figure 2.14 note the input similarities to the previous circuits. The two inputs enter the organelle at top left, the signal is routed to the gate, and the output exits the organelle on the bottom polysilicon line at the left. Also

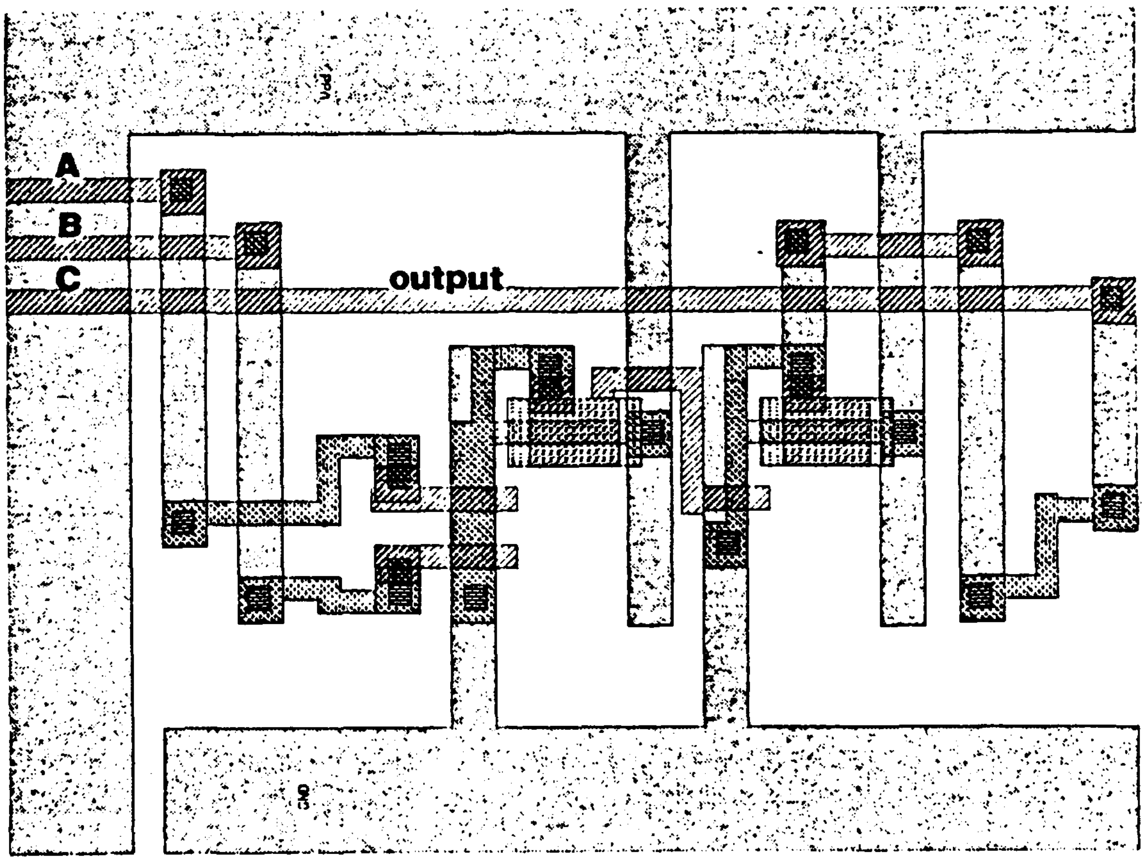


Figure 2.14 Data Path AND Gate

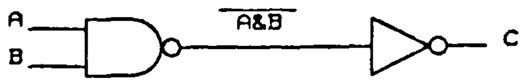


Figure 2.15 Gate Equivalent of Data Path AND Gate

note the difference among layouts of the MacPitts NAND gate and the MacPitts NOR gate, and the corresponding Mead-Conway cells [Ref. 4:p. 17].

7. A Three Input AND Structure In The Data Path

The three input AND was expected to produce gates similar to those of the two input AND, a series of cascaded NAND gates each followed by an inverter. Figure 2.16 shows the algorithm for the three input AND circuit, and Figure 2.17 depicts the resulting layout. The circuit is the equivalent of three ANDs due to associativity of AND.

8. Data Path Basic Organelles

When a MacPitts source algorithm is invoked by the linked binary MacPitts image by issuing the command

```
macpitts <filename> <options>
```

LISP object code is generated (unless the noobj option is specified, in which case MacPitts searches for a previously-created object file of <filename>.obj). In the filename.obj file it is observed that the data path logical operations are all derived from NOT, NAND, and NOR LISP operations. This is due to the fundamental hardware building blocks of MacPitts data path combinational logic being two input NAND and NOR gates, and NOT gates (inverters). Knowing this, the reason for the two-input gate implementation as depicted in the previous figures becomes clear.

Any data path logic organelle is composed of these primitives. The OR organelle is a NOR gate with an inverter on its output. The AND organelle is a NAND gate with an inverter on its output. In the data path, these organelles are assembled into macros in the organelles.lisp file of the MacPitts source code. The process of silicon compilation is thereby shortened, since some of the constituent parts are already put together.

A two input data path NAND gate chip is implemented exactly as it is specified. A three input NAND structure is implemented as expected, by cascading two NAND orgenelles (the three input NAND structure is not functionally equivalent to a three input NAND gate). The output, again, is what the LISP parenthesized notation would lead one to expect.

9. Bit Slice Combinational Logic

So far, all examples given have used inputs having one bit,but the data type specification for data path combinational logic is integer. Word size data inputs are treated in the expected way. Figure 2.16 illustrates a routine which performs the logical AND on two input vectors each four bits wide. Notice the similarity of this MacPitts program to those already given. The only differences between this routine and the AND of two bits are the PORT statements, which make logical and connective assignments between i/o ports and inter-chip hardware blocks.

```
;3AND.MAC
;SOURCE CODE FOR 3 INPUT DATA PATH <AND> GATE
(program 3and  1
        (def 1 ground)
        (def a port  input (2))
        (def b port  input (3))
        (def c port  input (4))
        (def d port  output (5))
        (def 6 phia)
        (def 7 phib)
        (def 8 phic)
        (def 9 power)
        (always
        (setq d (word-and (word-and a b)  c ))))
```

Figure 2.16 3and.mac

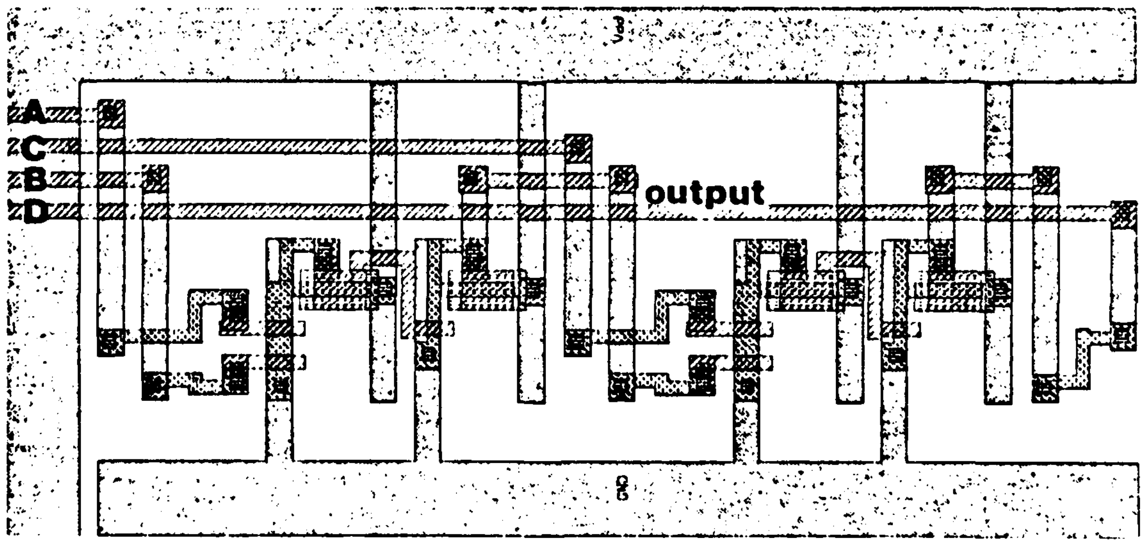


Figure 2.17 Circuitry from 3and.cif

Figure 2.18 illustrates the data path circuitry which implements this logic. It is evident that the logic is performed by replications of the fundamental MacPitts AND organelle, a NAND gate with inverted output. In comparing this circuit to Figure 2.14 the similarity becomes clear. The word-and integer operation as specified in the source algorithm translates to a data path AND organelle in the LISP object file. This organelle is replicated, instantiated, and connected to inputs and outputs to create the circuit (cifplot) shown in Figure 2.19. This data path word operation capability would not usually be applied to bit-width combinational logic, as the previous discussions might suggest, but rather to bit-slice operations such as word masking, parity checks, arithmetic operations, and so on.

10. Two Data Path Chips: Counters

A four bit resettable up-counter chip was designed by MacPitts using an algorithm given in the MacPitts documentation. Figure 2.20 shows the algorithm to specify the counter's behavior, and Figure 2.21 shows the resulting chip layout diagram. This example gives an indication of the implicative nature of MacPitts, which is actually a function of the LISP object code. There is a bank of three vertical drivers below the data path block in Figure 2.21. These are clock drivers, which drive the three phase clock.

```
;MULTIAND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<AND>  FUNCTION BY MACPITTS SILICON COMPILER
(program multiand 4
        (def 1 ground)
        (def a port  input (2 3 4 5))
        (def b port input (6 7 8 9))
        (def c port  output (10 11 12 13))
        (def 14 phia)
        (def 15 phib)
        (def 16 phic)
        (def 17 power)
        (always
        (setq c ( word-and  a   b   )   )   )   )
```

Figure 2.18 Multiand.mac

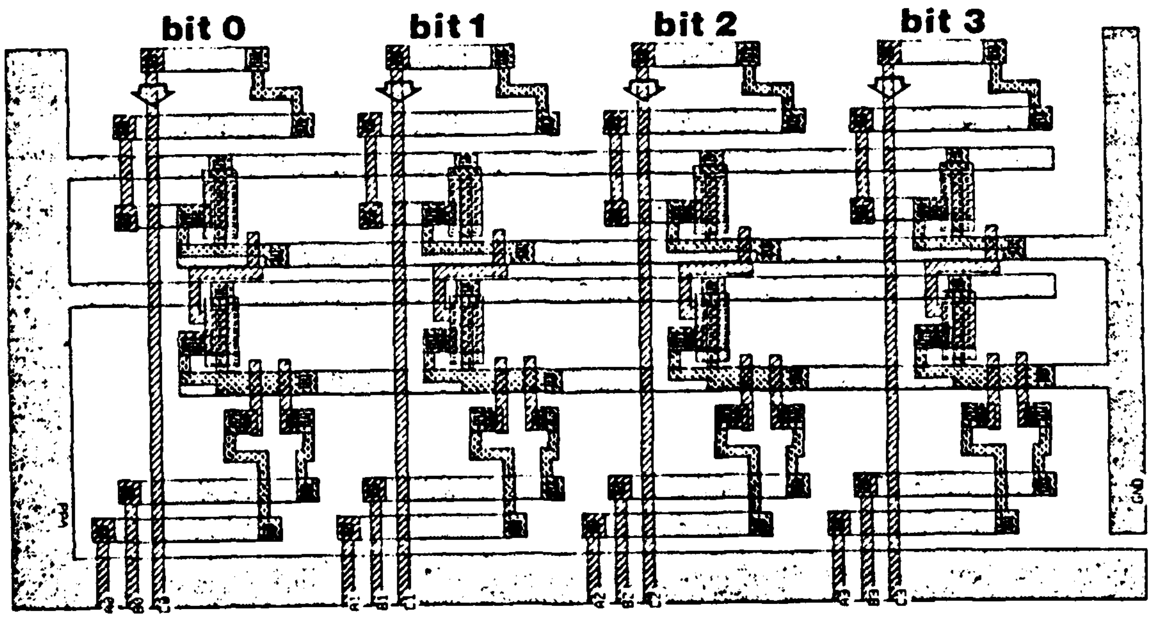


Figure 2.19 Logic Circuitry From Multiand.cif

```
;Example of MACPITTS algorithm to create a 4 bit counter
;illustrates use of "always" and "cond" commands
;title: count4.mac
(program count4  4
(def 16 power)
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def rst signal input 5)
(def count register)
(def cnt_up signal input 6)
(def ld_zero signal input 7)
(def out port output (12 13 14 15) )
(always
      (cond
            (ld_zero
                    (setq count Ø) )
            (cnt_up
                    (setq count (1+ count))  )    )
      (setq out count) )   )
```

Figure 2.20 Count4.mac

They connect to the clock lines on the bottom and to the count registers at the top.

There is a small Weinberger array beneath the clock drivers. A Weinberger array [Ref. 8] is used by MacPitts to control data path operations. It can be inferred from the size comparison between the data path block and the control block that this is a data intensive chip. The MacPitts algorithm reflects this, with many data operations such as SETQ and (1+ count), the increment statement, and few control operations such as

(cond(< conditional > <actions> ...)

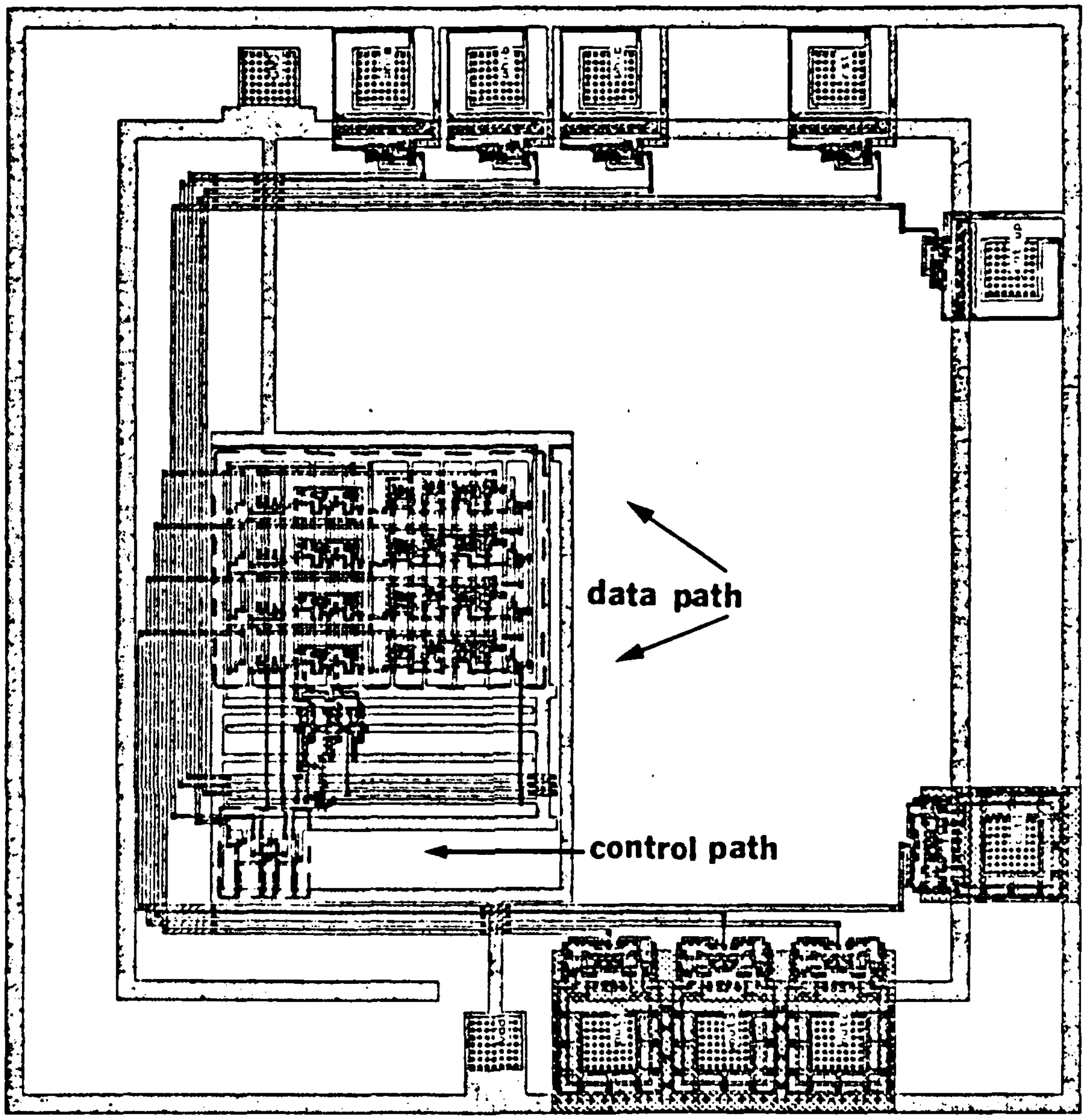


Figure 2.21 Count4.cif

where each < conditional > requires a decision. This decision making is perhaps more obvious in the generated object file, where each COND statement is translated to an IF statement. MacPitts implements the decisions more along the lines of a Pascal CASE construction than as an IF construction (the compiled LISP code reflects the IF logical testing, but it is set within a parallelizing command).

The SETQ form has operated on just ports so far. In count4.mac, the SETQ form operates on a register (COUNT, the current counter value). The last line in the algorithm, (setq out count), sets the output port to the current count register value. From the hardware perspective, this can be viewed as a latching or storage of the register contents, and clocking the contents to an output port. This is necessary in MacPitts since ports cannot store data. Only registers can store data in the data path, and MacPitts implements registers as master-slave flip flops.

The chips considered so far, with the exception of count4.mac, have been pure data path chips. In almost all useful chips, there will be a data path which is controlled by a Weinberger array control path. It is difficult to guess the relative sizes of the data path and control path from just the MacPitts driver algorithm. Nevertheless, if few conditional decisions are to be made and many arithmetic or logical operations are to be performed, the data path is likely to be the larger.

Figure 2.22 shows the algorithm (the .mac file) for count16ud.mac, the MacPitts driver for a 16 bit up/down counter. The signal and register names are self explanatory. The previous four bit up-counter was the prototype for this 16 bit up/down counter. The differences are in word length, the addition of a new input signal (count_down), the conditional test of count_down, and the decrement operation (1- count) if count_down is asserted true. It is usually a good idea to model a desired algorithm with a simpler prototype (functionally similar but having fewer inputs and outputs), and to test the prototype in the MacPitts command interpreter. For example, designing a four bit up counter is a good preliminary step when a 16 bit up/down counter is desired.

It can be inferred that the ratio of data path to control path size will be greater for this chip than for count4.mac. Figure 2.23 shows the resulting cifplot of count16ud.mac, and the 16 bit wide data path is indeed much larger than the control path, and as expected, much larger than the four bit counter data path also.

```
;Example of MACPITTS algorithm to create a 16 bit up/down counter
;copiously commented for clarity's sake
;title: count16ud.mac
(program count16ud   16
;note that the 16 opposite the title determines # of outputs
;doc. says data paths; actually equates to output pads(NOT paths)
;following 5 lines necessary every pgm:
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def 25 power)
;the counter will require a 16 bit width storage register(McP= m/s FF)
;... a count up enable signal,
;... a count down enable signal,
;... and a reset signal. These are described syntactically below:
(def rst signal input 5)
;this declares a  bank of 16 clocked  m/s FFs (see stippleplot)
(def count register)
(def cnt_up signal input 6)
(def cnt_dn signal input 7)
(def ld_zero signal input 8)
;the 16 output pads are specified:
(def out port output (9 1Ø 11 12 13 14 15 16 17 18 19 2Ø 21 22 23 24) )
;always command means to execute what follows every clock cycle
(always
;the cond (-ition) statement means to check the following guard
;conditions, and execute ONLY that one which is .true.
;execution of one guard precludes execution of any subsequent guards.
      (cond
;there are three guards to check:is ld_zero .true.?
;if not, is cnt_up .true.?
;if not, is cnt_dn .true.?
;if neither is .true. then exit the loop
            (ld_zero
;if ld_zero is asserted (high), then make count=Ø (i.e.,clr FFs)
                    (setq count Ø) )
            (cnt_up
;if cnt_up is asserted (high), then increment the count FF bank
                    (setq count (1+ count))  )
;if cnt_dn is asserted (high), then decrement the count FF bank
            (cnt_dn
                    (setq count (1- count))  )       )
;regardless of which (if any) operation is done, the FF contents
;are assigned to the output with the setq command.
      (setq out count) )   )
```

Figure 2.22 Count16ud.mac

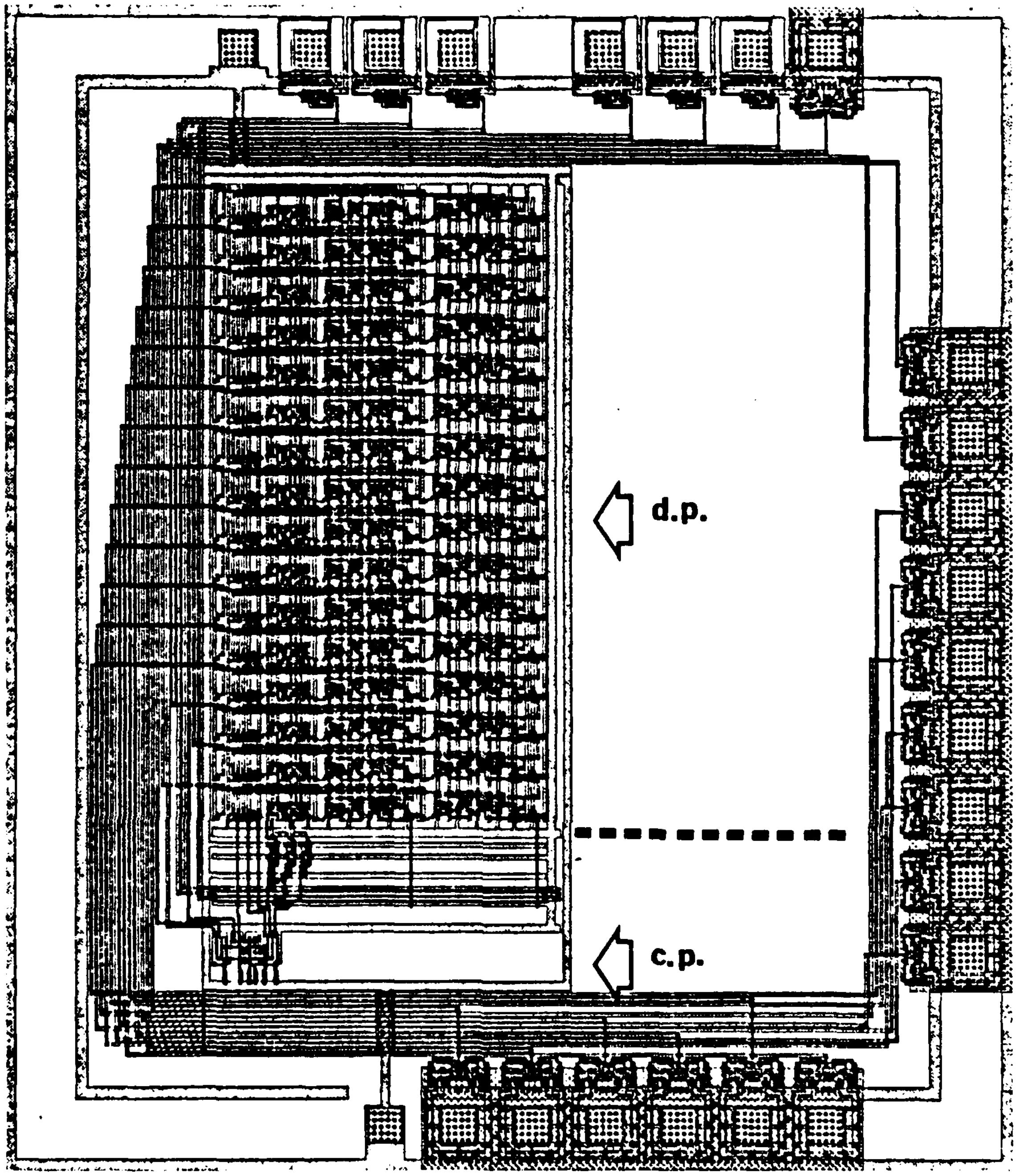


Figure 2.23 Count16ud.cif

B. COMBINATIONAL LOGIC STRUCTURES IN THE CONTROL PATH

The implementation of combinational logic in the control path of a MacPitts design is fundamentally different from its implementation in the data path.

In the data path, all combinational logic is constructed from basic two input NOR, NAND, and NOT cells, as described in the MacPitts source code file data-path.lisp. Any logical implementation, however complicated, is constructed from these three organelles (other organelles do exist in the organelles.l file, but they all are constituted either from these basic cells or permutations of these cells).

Furthermore, the specifications required by MacPitts in the data path are more oriented towards structure than behavior. For instance, when the programmer/designer writes the following algorithmic fragment

```
(word-and a(word-and b c))
```

what is being explicitly specified is a two-level gate structure. The innermost level comprises a two-input AND gate, the output of which is fed to the input of the second level AND gate, in parallel with the third input. Note that a single gate with more than two inputs is not permitted in the data path. The syntax constraints of the MacPitts compiled object code determine this structure. Again, this apparent limitation is not really a limitation at all because MacPitts is so constructed as to force decisions to be made in the control path. Consequently, the necessity of Boolean algebraic reduction in the data path combinational logic is highly unlikely.

1. Control Path Combinational Logic

The control path implementation of combinational logic is simpler than the data path implementation in two ways. It is behavior oriented, rather than structure oriented. The MacPitts designer needs only to specify the MacPitts LISP-like behavior of the structure, and the MacPitts environment produces a realization of it. This requires little (if any) Boolean reduction which might be required for complicated data path logical structures.

The control path combinational logic is also simpler structurally, in that it is always implemented in a highly-regular Weinberger array. A tradeoff between simplicity of layout and maximum circuit speed exists, however, and this topic will be considered in Chapters IV and V. Although a Weinberger array is geometrically simpler than a Programmable Logic Array (PLA), it is not as fast or as small.

The selection of which path is to perform the combinational logic is inherent in the MacPitts (the language) syntax. If the logical operator is a Boolean form and its antecedents are signals or flags, the control path will do the logic. If the logical operator is an integer form and its antecedents are ports or registers, then the combinational logic will take place in the data path. Thus, the syntax drives the selection of where the combinational logic occurs.

The initial MacPitts documentation offered some insight into these distinctions. A variety of tests were devised in the current investigation to explore the combinational logic implementation differences between the data and control paths. The experiments designed to arrive at the above conclusions for the control path logic are presented in the following sections.

2. <u>A Control Path AND Gate, And Control Path Syntax</u>

Casand.mac (cascaded AND gates, Figure 2.24) was the algorithm to create the initial structure to explore combinational logic implementation in the control path. The control path implementation of combinational logic requires a different kind of input declaration than does the data path. In the control path, the inputs must be declared as

<name> signal input <pin number>

This has the effect of coercion to Boolean (true or false, as opposed to one and zero) in the MacPitts environment.

Consequently, a different type of logical operator is required in the SETQ argument forms. In the data path, using defined-integer ports as inputs, the integer logic

SETQ forms are used (word-or, word-nand, etc). In the control path, however, Boolean SETQ forms are required (or, nand, etc.). The data path integer SETQ forms are limited to two logical arguments, whereas the control path SETQ forms are effectively unlimited as to number of logical arguments. This seemingly arbitrary constraint becomes understandable in view of structural implementation in the respective paths. In the data path, all logic must be implemented by cascades of two input gates. In the control path, all logic is implemented by a Weinberger array, which has no practical limit (except speed, pin count, and chip size) on the number of inputs.

Furthermore, the data path combinational logic restrictions are less strict (structurally speaking) than are the control path logical structures. For instance, in the data path all combinational logic structures are derived from NAND, NOR, and NOT gates, and implemented as macro organelles. In the control path, however, all logic structures are constrained to be NOR gates. The basename.obj file that results from a basename.mac file indicates all control path combinational logic implemented as NOR operations. Figure 2.25, casand.obj, shows the NOR function used to perform the AND function in the control path. All control path combinational logic operations are implemented in this fashion, as in the more common PLA.

```
;CASAND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<and>  FUNCTION BY MACPITTS SILICON COMPILER//2  input gate//
(program casand 1
        (def 1 ground)
        (def a signal input 5)
        (def b signal input 6)
        (def c signal output 7)
        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 8 power)
        (always
        (cond  (a
                 (setq  c (and a b)  ) )
               (b
                 (setq  c (and a b) ) )  ) )  )
```

Figure 2.24 Casand.mac

```
(((destination c)
  (source a)
  (source b)
  (logo casand)
  (word-length 1)
  (ground 1)
  (signal a input 5)
  (signal b input 6)
  (signal c output 7)
  (phia 2)
  (phib 3)
  (phic 4)
  (power 8))
 nil
 nil
 (((signal-output c) (nor ((primitive (gate 4)))))
  ((gate 4) (nor ((primitive (gate 3)) (primitive (gate 2)))))
  ((gate 3)
   (nor
    ((primitive (gate 1)) (primitive (gate Ø)) (primitive (signal-input a)))
  ((gate 2) (nor ((primitive (gate 1)) (primitive (gate Ø)))))
  ((gate 1) (nor ((primitive (signal-input a)))))
  ((gate Ø) (nor ((primitive (signal-input b))))))
 ((4 (phic))
  (3 (phib))
  (2 (phia))
  (1 (ground))
  (8 (power))
  (6 (input b (signal-input b)))
  (5 (input a (signal-input a)))
  (7 (output8 c (signal-output c)))))
```

Figure 2.25 Casand.obj

The AND plane in an NMOS PLA is actually comprised of NOR gates, its function is logical AND, but its constituent circuits are NOR gates. The NOR structure which the control path uses is also different topologically from that used in the data path.

A concise review of the data path and control path variable types illustrates the usage differences:

	DATA TYPE	
	BOOLEAN (true,false)	INTEGER (word valued)
STORAGE ELEMENT	flag	register
NON-STORAGE ELEMENT	signal (input,internal)	port (all types)

All storage elements are implemented as master-slave flip-flops. They retain their value until a new value is clocked into them. The flags are one bit wide, and are two-state devices, either true or false. The registers have a capacity of the data path width as declared in the initial PROGRAM statement in the MacPitts source program written by the chip designer.

Non-storage elements are used primarily for data communication within a clock cycle, where clock cycle here is taken to refer to the command interpreter clock cycle, and not one of the three off-chip clock phases which a MacPitts design requires. The determination of the value of these non-storage elements is germane to pipelined digital machines. When used in any application, care must be taken so that their value is the one necessary for subsequent stages of logic. A thorough understanding of the counter-intuitive parallelism inherent in MacPitts (the language) is necessary to avoid mistakes here. MacPitts is not like the standard sequentially executed higher level languages. There are at least three levels of implicit parallelism possible in a MacPitts algorithm, and an understanding of parallel operations is necessary to avoid functional errors. This consideration is germane to MacPitts programming, and will be considered in detail later in this Chapter and in Chapters III and IV.

The next-to-last line in Figure 2.24 illustrates a conditional. The (b ...statement is a checked <condition> argument of the beginning COND (do upon condition) statement, as is (a... . If condition a is false, and condition b is false, then no output is SETQ'd. Intuition would suggest that the output would then either remain at its last value or transition to tristate, neither of which is correct. The output is pulled low by the Weinberger array circuitry. This is evident in Figure 2.27 the Weinberger array from casand.mac, and in Figure 2.28, the logic gate equivalent. The (cond (t ... form can be used to set a desired output, but it is usually better suited as a default conditional.

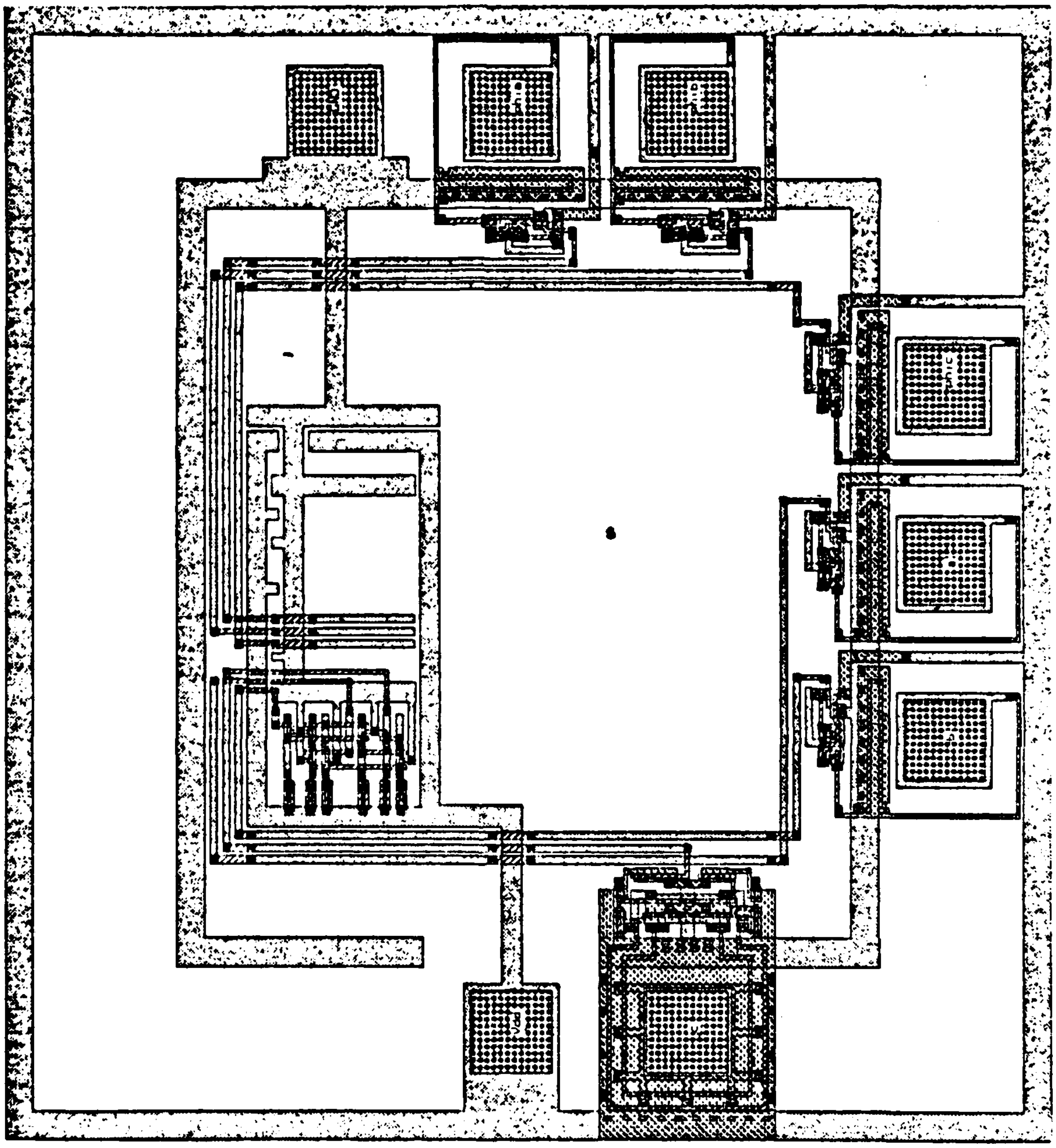


Figure 2.26 Casand.cif

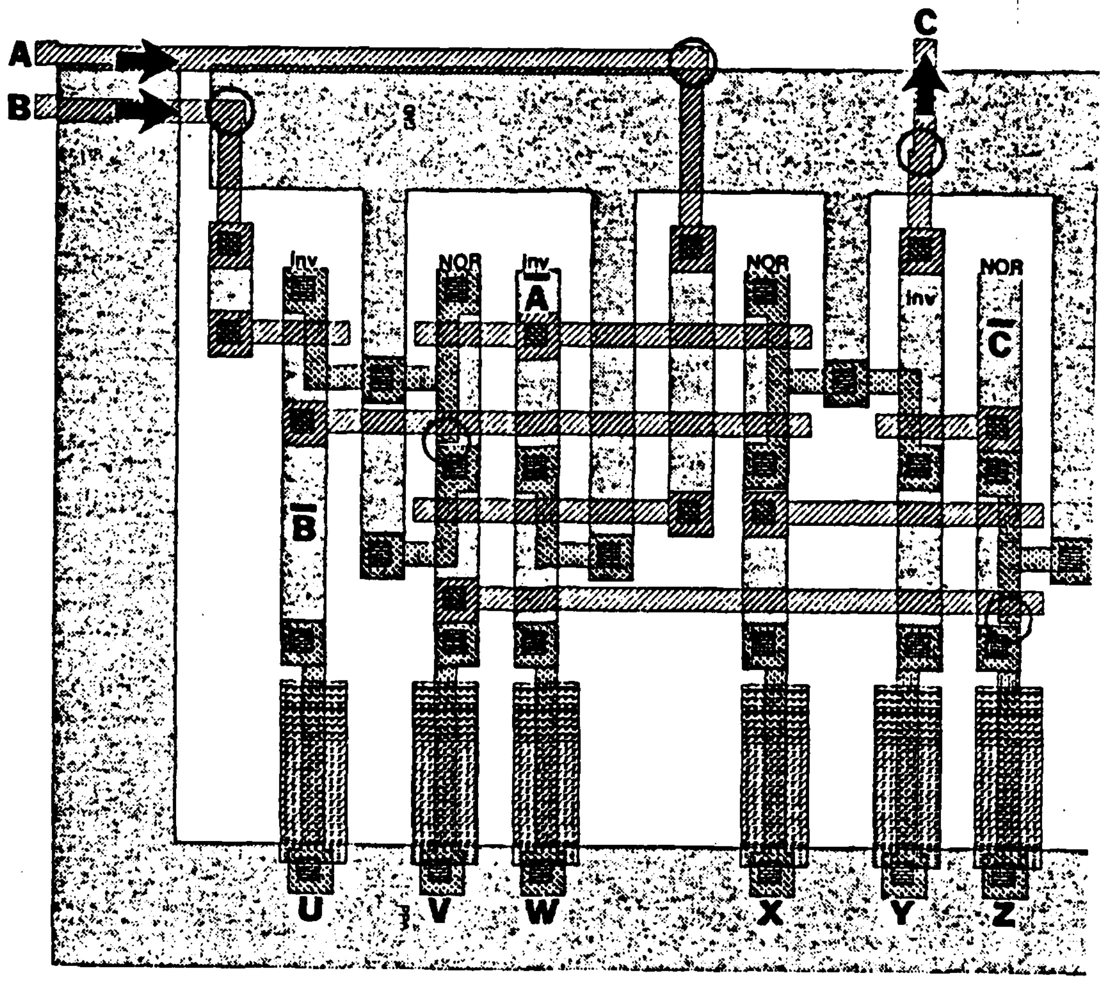


Figure 2.27 Casand.cif Weinberger Array

MacPitts does not view this algorithm as a usual high level sequential test, however, but rather as a parallel test of a and b. The non-intuitive parallelism of MacPitts was mentioned in the previous paragraph, as was the similarity of the MacPitts COND statement to the Pascal CASE statement. Some elaboration will serve to clarify this necessary concept. MacPitts evaluates all of the forms within the scope of a COND statement in parallel, in a mutually exclusive fashion. With regard to mutual exclusivity, it is then similar to the CASE statement; each condition under the scope of a COND can be modelled as a flow-of-control switch, either turning on the evaluation of its constituent forms or else skipping over their evaluation. The analogy does not hold further than this, however, because MacPitts evaluates all of the conditions under a COND in parallel. The object code created from a MacPitts source file illustrates this well. An example is

```
  (cond
     (hot       (setq fan_on  t))
     (cold      (setq fan_on  f))
     (ok        (setq fan_on  f))   )
```

Where hot, cold, and ok are Boolean variables (signals or flags), fan_on is in this case a Boolean signal output which is to be turned on (t) or off (f) depending on an input temperature signal. COND forces parallel evaluation of these three conditions under its scope, hot, cold, or ok.

The last parenthesis in this fragment closes the beginning parenthesis prior to the COND, bringing the three conditions under its scope. Since these conditions are evaluated in parallel, a better code fragment would be

```
(cond
   (hot      (setq fan_on  t))
   (cold     (setq fan_on  f))
   (t        (setq fan_on  f))   )
```

where the last line indicates TRUE, i.e., it is always true. Since COND evaluates in parallel with mutual exclusion based upon order, if either of the first two conditions is true, then the remaining conditions are not evaluated. If neither of the first two conditions is true, however, then the fan will be turned off. This code fragment permits one less signal input (or one less flag used) on the chip, and use of the TRUE t condition should always be considered. Its use is not necessary, as indicated by the first code fragment.

MacPitts produces an accompanying object code which structurally resembles the following fragment

```
(if
   (par(hot ...    )
       (cold...    )
       (t   ...    )  )   )
```

where the COND translates to an IF, and the parallelism of MacPitts is evident in the PAR (parallelize) embracing the three constituent conditions under the COND. Parentheses are as important in MacPitts as thev are in LISP. In the last line above, there are three closing parentheses. The innermost closes the TRUE condition, the middle parenthesis closes the PAR (parallization of condition checking), and the outermost closes the IF (cond) statement.

The LISP object file of casand.mac in Figure 2.25 indicates the LISP equivalent of the MacPitts (language) algorithm, and shows how LISP views the NOR gate inputs as primitives. MacPitts is also able to compile a chip layout directly from a LISP object code. This is an option for the designer who is fluent in LISP in that customizing of the code and hence the chip's structure is possible. RVLSI-3 [Ref. 6:p. 4] describes how to create a chip design from an existing LISP object file.

Figure 2.26 shows the chip resulting from casand.obj. The pads are all placed clockwise around the periphery of the chip in the order specified in the .mac file (Figure 2.24). This built-in function of MacPitts lends itself to both errors and possibilities of improvement. It is easy to identify pad function if the MacPitts algorithmic source file (written by the designer) is at hand.

Figure 2.27 also shows the topological difference between the data path and the control path. In previous data path circuits, all combinational logic was implemented with recognizable NMOS logic gates. In the data path, the Weinberger array is made up of many vertical metal columns with perpendicular polysilicon lines cutting across them. Figure 2.28 illustrates the structure more clearly.

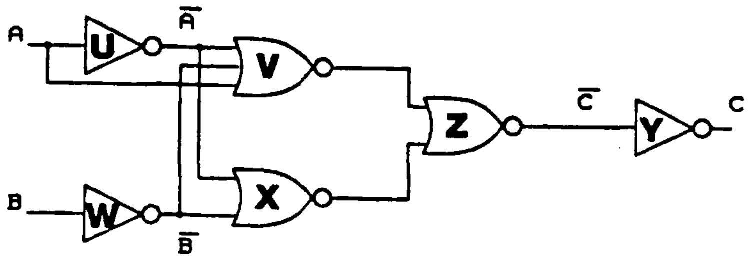


Figure 2.28 Gate Equivalent of Casand Weinberger Array

In Figure 2.26 the Vdd input rail did not connect with the main Vdd bus (it has been corrected in Figure 2.26). It passes through the polysilicon vias and stops abruptly. The reason for this is the expectation of minimum chip size which MacPitts harbors. For any but the simplest of chips, the Vdd comb will extend out to the input Vdd rail. If it does not, the Vdd pad can be placed almost at will by modifying its position in the basename.mac file. RVLSI-3 discusses this [Ref. 6:pp. 11-13]. The designer can exercise a fair amount of latitude in pad placement, and MacPitts will accommodate most of the time. The suggestion in RVLSI-3 that GND be placed near the beginning and Vdd be placed near the end is a good one. The main problem here would arise if GND were placed on the right side of the chip so that it contacted the Vdd comb (which it will do if care is not exercised). MacPitts places pads exactly in the order specified, and does no pad functional error-checking. Similarly, if a pad is dual-defined, MacPitts permits it with no diagnostics. This extends to the same pad being used for both Vdd and an input signal. So care is important in both pad specification and positioning.

There exists the possibility for some improvement in chip speed by designer intervention in specifying the pad location. By moving pads five, six, and seven (input and output signals in casand.mac) closer to the Weinberger array, the metal run lengths can be reduced and thus the metal to substrate capacitance. This results in a somewhat faster chip, all other factors being equal.

Figure 2.27 is a blowup of the Weinberger array generated by casand.mac, and Figure 2.28 is its logic gate equivalent. The Weinberger array is a versatile PLA-like structure generally used to implement sequential logic. In this chip, as an unclocked circuit, it implements a combinational function. Weinberger array gate instantiation errors were first detected here (circled). Note the two half lambda gaps in the NOR gate inputs. By Caesar editing, unexpanding of affected cells, and grep-searching the .cif files it was discovered that these errors occur whenever certain NOR gate inputs are invoked. The errors themselves were suspected to reside in the control.lisp file of the MacPitts source code. Two specific cells appear to generate these errors: partial-gate-input ('ground-right), and partial-gate-input ('ground left). Each is one-half lambda too short. Chapter VI will tre- the solution of this problem. The MacPitts command interpreter does not detect this type of error, since it only exercises the algorithm. Lyra or a similar design rule checker will detect this error. The designer would do well to visually note MacPitts' inherent errors and correct them prior to submission to a design rule checker (drc).

3. A Control Path OR Gate

Figure 2.29 illustrates the MacPitts algorithm to create a two input OR realization in the control path. The OR function is realized by a selective SETQ choosing process, in a similar fashion to the previous AND realization.

Figure 2.30 is the Weinberger array logical unit of casor.cif. The inputs are brought in on either side, and the output comes out from the middle of the structure. The same instantiation errors as in the previous chip were generated. Partial-gate-input (gnd left) is depicted in the upper left of the stipple plot, and partial-gate-input (gnd right) is depicted in the lower right of the plot (circled) in Figure 2.30.

The logical operation of the Weinberger array could stand some clarification. Figure 2.31 depicts a gate-level functional representation of Figure 2.30, the control path

```
;CASOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<or>  FUNCTION BY MACPITTS SILICON COMPILER//n  input gate//
(program casor 1
        (def 1 ground)

        (def a signal input 5)
        (def b signal input 6)
        (def c signal output 7)

        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 8 power)

        (always
        (cond  (a
                 (setq  c  t) )
               (b
                 (setq  c  t) )  )     )        )         )
```

Figure 2.29 Casor.mac

implementation of a two input COND-test OR structure. Looking at Figures 2.30 and 2.31 and 2.28, the function will be explained. Figure 2.30 has four depletion mode transistors (control columns to MacPitts). The left most transistor is the first inverter in Figure 2.31. The next column in Figure 2.30 serves as the top NOR gate in Figure 2.31. Moving right in Figure 2.30, the next column is the output inverter. And the rightmost column is the lower NOR gate corresponding to Figure 2.31. When viewed as a gate level equivalent, it can be seen that the Weinberger arrary is both larger and slower than its data path equivalent (cf. Figure 2.6). In the control path, the signal requires approximately four gate delays to propagate from input to output. This slowness has been somewhat mitigated by

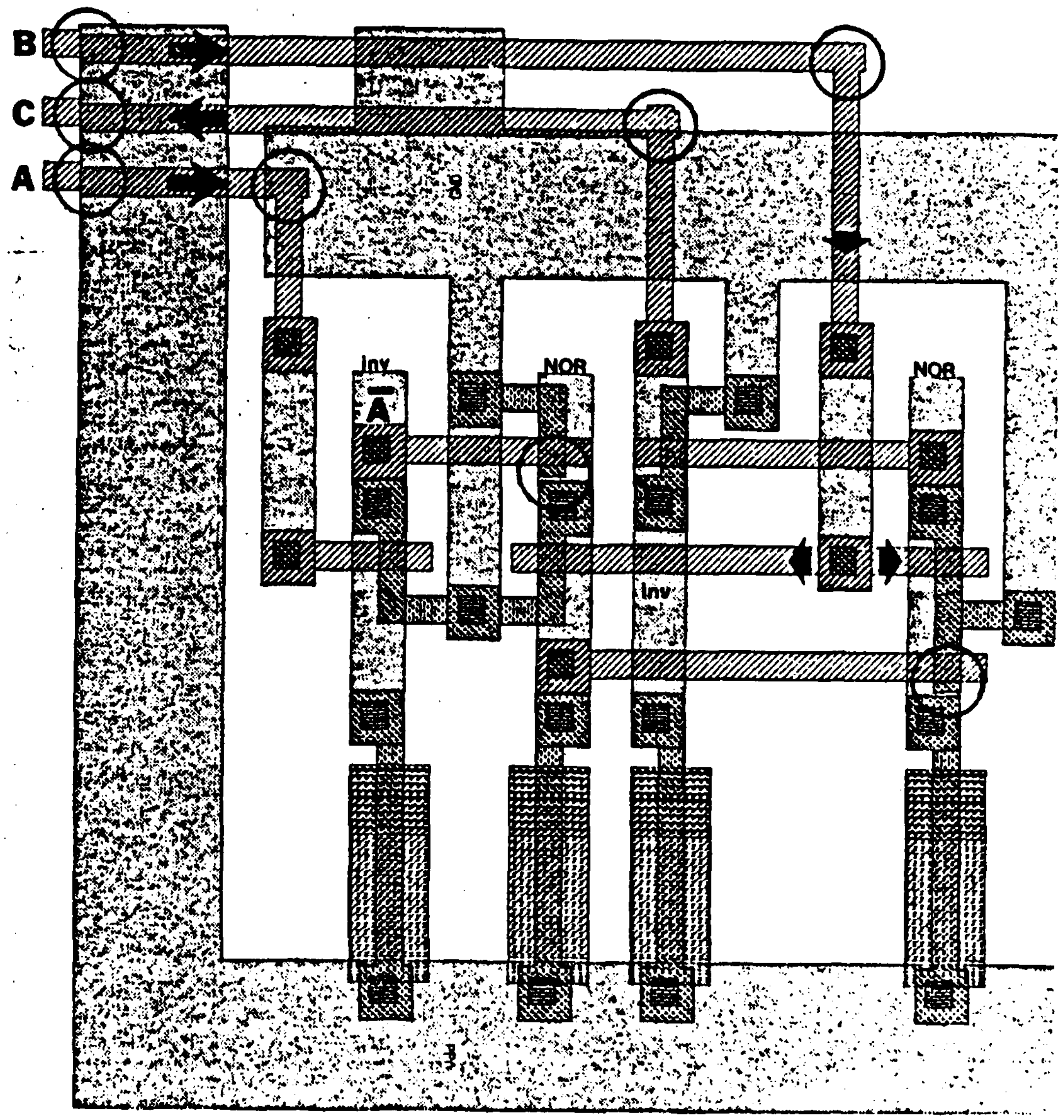


Figure 2.30 Casor Weinberger Array

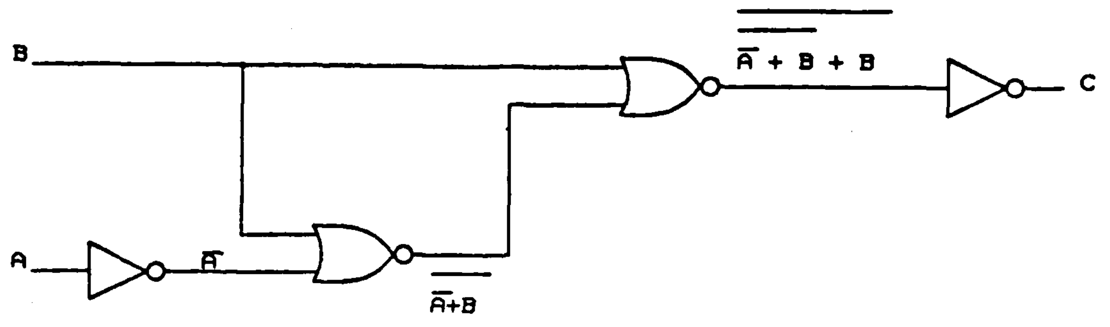

B
$\overline{\overline{\overline{A}+B}+B}$
C
A
$\overline{A}$
$\overline{\overline{A}+B}$

Figure 2.31 Gate Equivalent of Casor Logic

the large aspect ratio of the pupllup transistors (bottom, Figure 2.30). The comparable logic gate in the data path only requires approximately two gate delays, one for the NOR gate and one for its subsequent inverter (Figure 2.7).

This simple COND-driven control path OR gate serves as an indication of how MacPitts constructs similar yet more complicated Weinberger Array structures. The decision logic is quite unlike that of a PLA. In a standard NMOS AND plane-OR plane PLA, a signal *may* experience at *most* four gate delays (considering input and output inverters both active, and pass transistors inducing a very small time delay). For this simple OR circuit, a gate delay of approximately four is realized. The cascading of NOR and inverters induces even more delay for more complicated Weinberger array circuitry.

4. <u>A Four Input OR Gate In The Control Path</u>

A quad-input OR structure is specified algorithmically in Figure 2.32. The OR logic which is implicit in MacPitts specifications is perhaps clearer here than in the two input OR structure. The COND statement forces a Boolean test of each input, and selects the appropriate output. To reiterate, the COND statement and its attendant forms can be viewed as the if-then-else construct of many higher level languages. The difference is that MacPitts tests the condition forms in parallel, and not in a serial fashion as most higher level software compilers would. The mutual exclusivness of the <conditions> is determined by serial order, however, even though the testing of the conditionals is done in one clock cycle (or in parallel).

```
;QUADOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<or>  FUNCTION BY MACPITTS SILICON COMPILER//4  input gate//
(program quador 1
          (def 1 ground)

          (def a signal input 5)
          (def b signal input 6)
          (def c signal input 7)
          (def d signal input 8)
          (def e signal output 9)

          (def 2 phia)
          (def 3 phib)
          (def 4 phic)
          (def 1Ø power)

          (always
          (cond  (a
                   (setq  e  t) )
                 (b
                   (setq  e  t) )
                 (c
                   (setq  e  t) )
                 (d
                   (setq  e  t) )
                               )     )       )         )
```

Figure 2.32 Quador.mac

This is reflected in in the resulting structures. Figure 2.33 shows the labelled Weinberger array resulting from quador.mac, and Figure 2.34 is its logic gate equivalent. A strength of MacPitts is that it forces the designer to consider both behavior and structure while in the process of writing the driver algorithm. This is considered to be advantageous, inasmuch as the abstractness factor is minimal. There are two broad categories of silicon compilers, behavior oriented (e.g., MacPitts), and structure oriented (e.g., Bristle Blocks). In Bristle Blocks and most other register transfer logic (RTL) silicon compilers, a structure is the fundamental building block. The structures (register, adder, ALU, gate) must be connected appropriately to implement the desired behavior. In MacPitts, the desired behavior of the chip is the input to the silicon compiler and the chip which implements this behavior is the output. The experienced designer is aware of the structure that results from a given behavioral specification, and has the latitude to optimize the algorithm accordingly. This has been mentioned previously, regarding pad placement and COND. Optimization will be treated further later in this thesis.

5. A Four Input AND Gate In The Control Path

Figure 2.35 shows the algorithm to create a four input AND gate in the control path, and Figure 2.36 shows the Weinberger array from the logic block of quadand.cif.

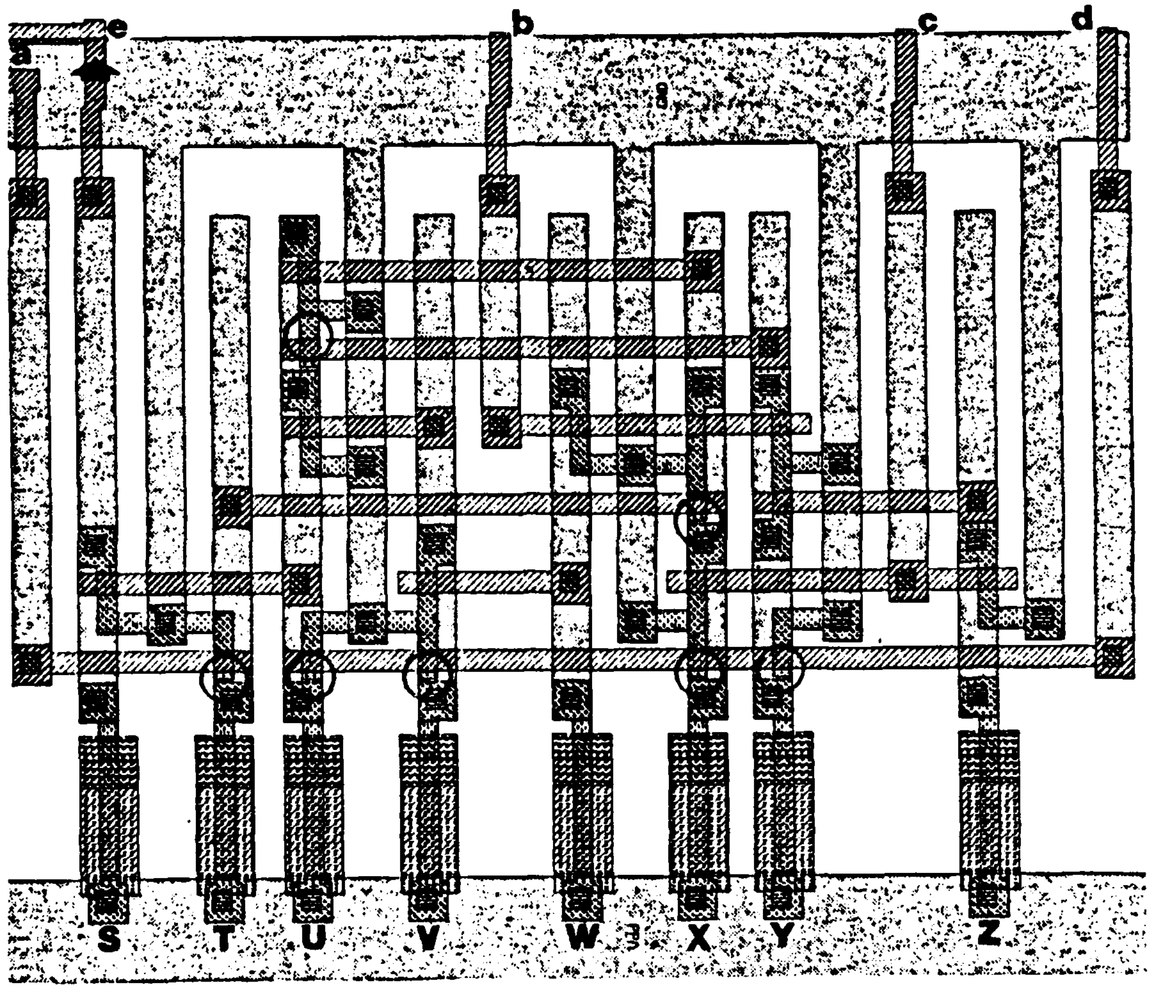


Figure 2.33 Quador Weinberger Array

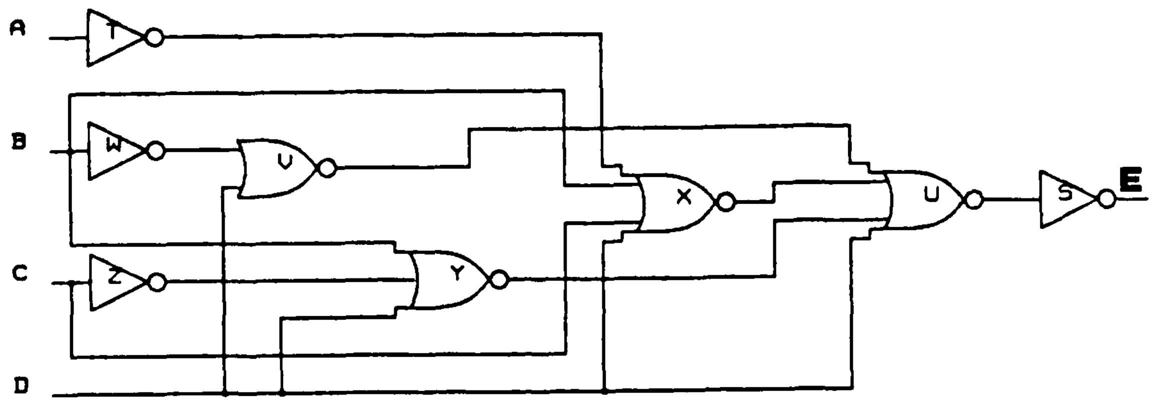


Figure 2.34 Gate Equivalent of Quador Logic

Note the errors generated in this simple four input, one output circuit (circled, Figure 2.36). There are seven gate gap errors (all partial-gate-inputs), and three alignment errors. The alignment errors are actually derived from mis-translation of the Weinberger array interface cell by MacPitts (the program). The interface cell is created with the proper pitch, set aside in the VAX 11/780's memory, then invoked and its image translated to the proper position in the upper-left of the Weinberger array. By convention, upper left on the MacPitts chips refers to the nominal position of the GND pad, position one.

```
;QUADAND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<and>  FUNCTION BY MACPITTS SILICON COMPILER//4  input gate//
(program quadand 1
        (def 1 ground)

        (def a signal input 5)
        (def b signal input 6)
        (def c signal input 7)
        (def d signal input 8)
        (def e signal output 9)

        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 1Ø power)

        (always
        (cond  (a
                 (setq  e (and a b c d)  ) )
               (b
                 (setq  e (and a b c d)  ) )
               (c
                 (setq  e (and a b c d)  ) )
               (d
                 (setq  e (and a b c d)  ) )
               (t
                 (setq  e f              ) )
                          )     )       )          )
```

Figure 2.35 Quadand.mac

So what appears to be three separate alignment errors is actually just one cell translation error. This error should be repairable in the macro-instantiation portion of MacPitts, although further investigation will consider also the possibility of an error in program installment.

6. A 15 Input OR Gate In The Control Path

It was stated previously that MacPitts will permit no more than five deep cascading of the same gate organelle in the data path. This is not the case, however, in the control path. Figure 2.37 shows a MacPitts algorithm to create a 16 input OR circuit. Note again how natural the specification is, and the intuition it gives into both behavior and structure. To reiterate: in the data path, one specifies structure explicitly and the implicit behavior results. In the control path, one specifies behavior explicitly, and the implied structure (always a Weinberger array) results (cf. Figure 2.13, data path AND, Figure 2.25, control path AND). The suggestion is to specify as much combinational logic as possible in the control path (this decision fortunately never arises because MacPitts is not primarily a combinational logic design tool).

In program multior.mac the data path width is still one. The data path width actually refers to the number of outputs from the chip (in the absence of a data path), not as its name would lead one to believe. So with one output, the data path width is one, even though there are 16 inputs.

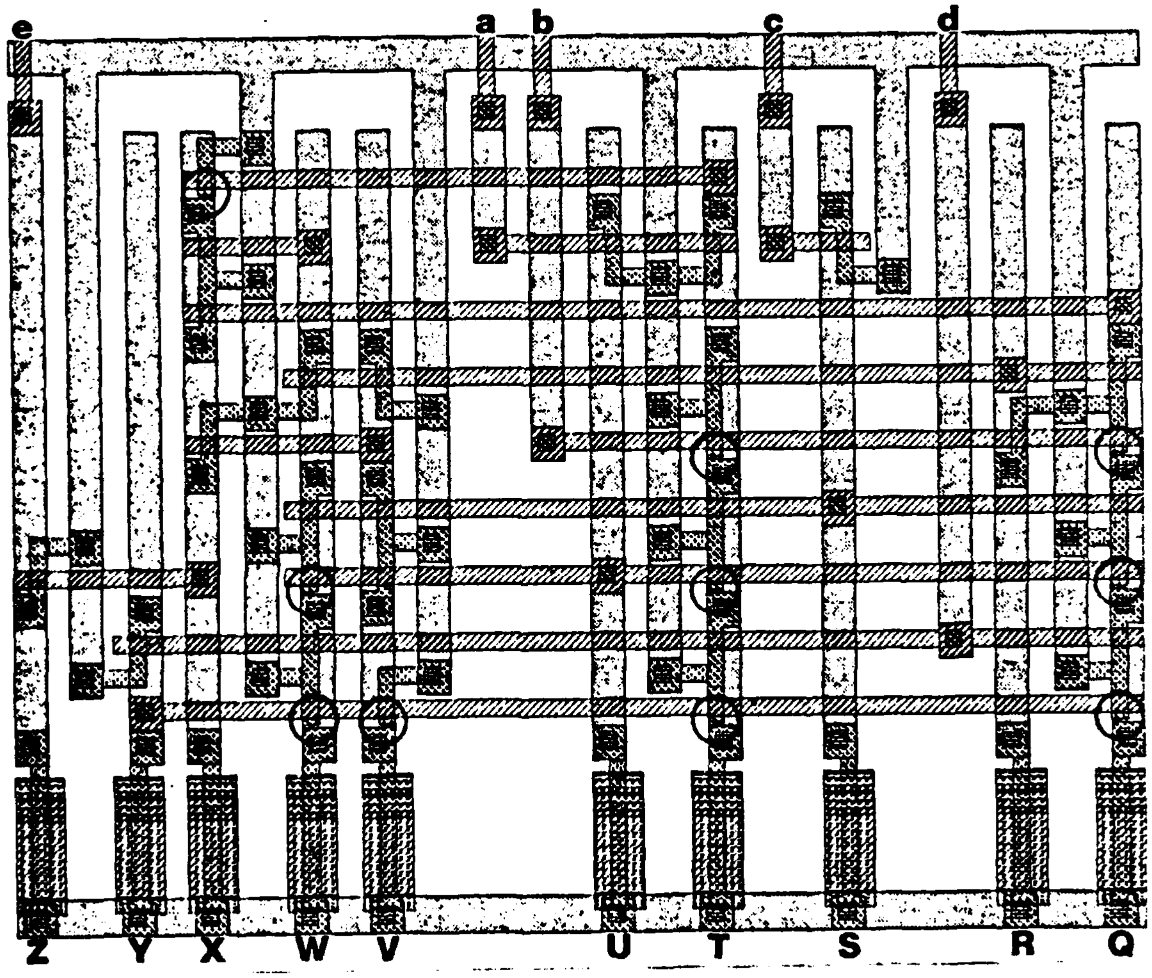


Figure 2.36 Quadand Weinberger Array

The format for data path width specification is

```
(program <program name> <data path width>
```

Figure 2.38 shows the chip structure of multior.cif. It is seen that the chip is composed of a small un-clocked control path unit alone, in the middle of the Weinberger Vdd/GND comb. There are no data path organelles. As previous experience would suggest, this control path has several instantiation gap errors and cell translation errors (see Figure 2.25). The large number of depletion pullup transistors inherent to the Weinberger array is also apparent. Combinational logic implementation in the control path typically requires more depletion pullups than would be required for the equivalent structure in the data path, because all control path logic is done with NOR gates. Since the pullups are always turned on, a MacPitts chip is not expected to be very conservative of power. In the four input OR gate, there were eight pullups in the Weinberger array, and seven instantiation gap errors. In the 16 input OR circuit, there are 30 pullup transistors, and approximately 40 gap errors. These errors are caused by instantiation of the partial-gate-input cells (specifically, partial-gate-input-ground-left and partial-gate-input-ground-right), and they occur every time one of these cells is called.

```
;MULTIOR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<or>  FUNCTION BY MACPITTS SILICON COMPILER//16  input gate//
(program multior 1
        (def 1 ground)
        (def 2 phia)(def 2 phib)(def 3 phic)
        (def a signal input 5)
        (def b signal input 6)
        (def c signal input 7)
        (def d signal input 8)
        (def e signal input 9)
        (def f signal input 10)
        (def g signal input 11)
        (def h signal input 12)
        (def i signal input 13)
        (def j signal input 14)
        (def k signal input 15)
        (def l signal input 16)
        (def m signal input 17)
        (def n signal input 18)
        (def o signal input 19)
        (def p signal input 20)
        (def q signal output 21)
        (def 22 power)
        (always
        (cond  (a
                 (setq  q t) )
               (b
                 (setq  q  t) )
               (c
                 (setq  q  t) )
               (d
                 (setq  q  t) )
               (e
                 (setq  q  t) )
               (f
                 (setq  q  t) )
               (g
                 (setq  q  t) )
               (h
                 (setq  q  t) )
               (i
                 (setq  q  t) )
               (j
                 (setq  q  t) )
               (k
                 (setq  q  t) )
               (l
                 (setq  q  t) )
               (m
                 (setq  q  t) )
               (n
                 (setq  q  t) )
               (o
                 (setq  q  t) )
               (p
                 (setq  q  t) )
                              )     )        )
```

Figure 2.37 Multior.mac

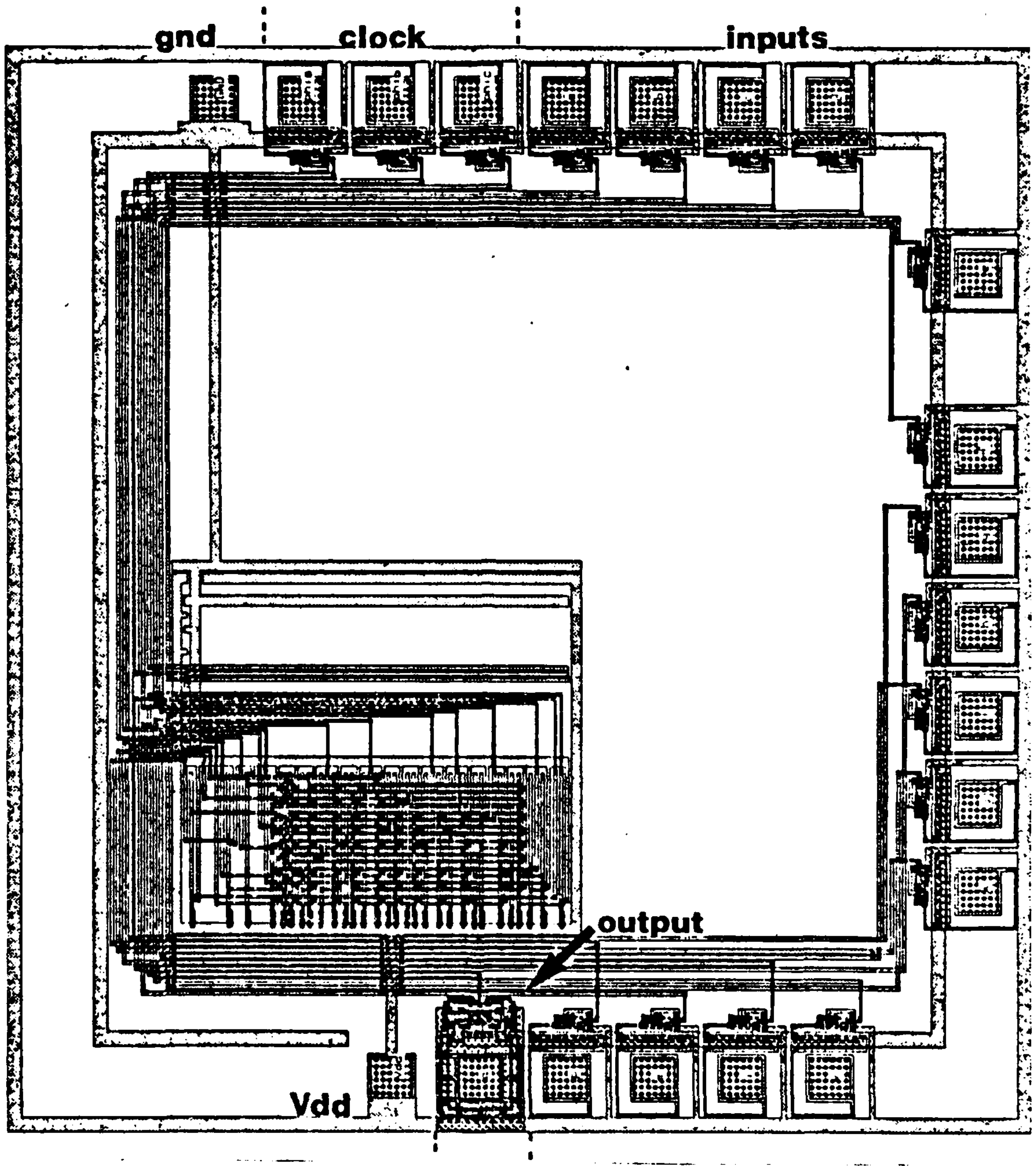


Figure 2.38 Multior.cif

MacPitts is limited in the data path as to how many combinational logic cascades may be made. Since the control path is designed to make decisions, the combinational logic cascading constraint is absent for most practical chips. Nevertheless, an error was detected in the multior.cif file, Figure 2.38. From multior.mac in Figure 2.37, one would expect the chip to have 22 pads, 16 input pads, one output pad, three clock pads, one ground pad, and one Vdd pad. The cifplot only shows 21 pads. This error does not show in the command interpreter. The 16 input OR function works as expected there. The error apparently lies elsewhere than in the .mac file. The chip does function nevertheless, but as a 15 input OR gate instead of as a 16 input OR gate. The pad deletion error (one fewer pads instantiated than specified in the .mac file) occurs whenever an OR gate having more than five inputs is specified in the .mac file. This is an unexpected error, though not very serious. The control path is rarely called on to do this sort of logic. If a special function of this type is required of a MacPitts chip, the designer can circumvent this problem by specifying an extra input pad in the .mac file. The chip will compile to cif, but the extra pad will not be instantiated nor will any of the attendant combinational logic or wires.

7. Control Path Semantics

The syntax (algorithm rules) for combinational logic in the control path has been illustrated in the previous sections. To gain an understanding of MacPitts, the semantics (what the algorithm means) is more important than how to say what it means.

The parallelism possible in MacPitts has been previously referred to in the discussion of parallel testing of conditions under a COND statement. This is not the only place where MacPitts forces parallelism. Parallelism is also forced upon all <actions> within a true condition under a COND. The general form of a COND statement is

```
(cond ( <condition>  <actions>  <transition>  ))
```

The <condition> is a Boolean variable upon which the true/false test is made, the <actions> are SETQs, and the <transition> is one of GO, CALL, or RETURN (to be discussed in Chapter IV). In the previous example, both hot and cold were Boolean conditional variables which would be tested in parallel. The <actions> under the COND refer to a set of SETQ assignment operators, and the SETQ's under a COND are all done in parallel, or simultaneously. The <transition> form indicates a state transition to be made if <condition> is evaluated as true. This state transition occurs in parallel (same clock cycle) with the <actions> associated SETQ's. The state transition mechanism of MacPitts is very straightforward and natural to a designer familiar with

Mealy type finite state machines. This topic will be considered in depth in Chapters IV and V.

Note the difference between the parallelism implied within the COND and that parallelism implied in condition evaluation. The conditions are all examined in parallel, and for the first one that evaluates to logical TRUE, all forms within its scope are executed in parallel. This high degree of implicit parallelism makes MacPitts ideally suited for pipelined architectures. Consider the following code in which three Boolean conditionals determine the outputs. The destinations of the SETQs are also Boolean, and in this case are non-storage elements (signals). The outputs are declared signals instead of flags (which are storage devices) so that when they are not set within a clock cycle they will transition to false.

```
(cond
   (hot
        (setq fan_on       t)
        (setq windows_open t)
        (setq doors-open   t)
        (setq heater_on    f))
   (cold
        (setq fan_on       f)
        (setq windows_open f)
        (setq doors_open   f)
        (setq heater_on    t))
   (t
        (setq windows_open t)
        (setq doors_open   t))   )
```

This algorithm models a simple digital home temperature controller where f refers to an inactive or closed device, t refers to an active or open device. and a comfortable temperature deadband exists between heating and cooling requirements. All three Boolean conditions (hot, cold, and true) are tested in parallel. The order of mutual exclusion is the order in which the conditions are written (if both cold and t are true simultaneously, only the actions under cold will be executed). The conditional (t... is the MacPitts equivalent of a reserved word, and indicates the always true conditional. It is used in this algorithm as the default state of the system, where the temperature is comfortable enough to leave both the doors and windows open. Even though (t... is always true, the evaluation order of the conditionals prevents the forms under its scope from being set unless both the preceeding conditionals are false. The actions under each true condition are also performed in parallel, or in the same clock cycle. So the testing of all three conditions and the resultant SETQ <actions> occur in only one clock cycle, due to the implicit parallelism of MacPitts. It is not necessary for the MacPitts programmer to explicitly parallelize the forms under a COND, the MacPitts compiler does this every time it encounters a COND. The (setq <output> f) statements under the hot and cold CONDs are not required for this system. As explained previously, the Weinberger array will set the output false if it is not explicitly driven true for non-storage Boolean variables. The (setq <output> f) statements have the advantage of added clarity in the MacPitts driver algorithm at the expense of increased size of the Weinberger array (more decisions are required).

The following code fragment produces the same results, though is somewhat more obscure:

```
(always
   (par
        (setq fan_on         hot)
        (setq heater_on      cold)
        (setq windows_open   (not cold))
        (setq doors_open     (not cold))))
```

In this example, no conditional testing is necessary although the results are equivalent to the previous example. On every clock cycle, all of the forms embraced by PAR are executed.. On each clock cycle, the fan, heater, windows, and doors are set to the correct state. The resulting hardware is simpler, since fewer decisions are required. This is the preferred format when conditional testing can be explicitly done with Boolean logic in the Weinberger array. But this code fragment lacks the ability to branch. When transfer of control is required, then it is necessary to use the full generalized COND statement

```
cond( <conditional> <actions> <transition> )
```

form instead of the truncated version

```
cond( <conditional> <actions> )
```

8. Five Input AND Gates In The Control Path

The savings of area in the Weinberger array can be substantial when Boolean decisions are made without a precedent COND statement. Figure 2.39 shows the MacPitts code used to generate a five input AND gate using COND for each output, and Figure 2.40 shows the resulting Weinberger array. Figure 2.41 is the logic gate equivalent of the five input COND driven AND gate.Contrast this with Figure 2.42 illustrating the code for generation

```
;FIVEAND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<and>  FUNCTION BY MACPITTS SILICON COMPILER//5  input gate//
(program fiveand 1
        (def 1 ground)

        (def a signal input 5)
        (def b signal input 6)
        (def c signal input 7)
        (def d signal input 8)
        (def e signal input 9)
        (def z signal output 1Ø)

        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 11 power)

        (always
        (cond  (a
                 (setq  z (and a b c d e)  ) )
               (b
                 (setq  z (and a b c d e)  ) )
               (c
                 (setq  z (and a b c d e)  ) )
               (d
                 (setq  z (and a b c d e)  ) )
               (e
                 (setq  z (and a b c d e)  ) )
               (t
                 (setq  z   f              ) )
                           )     )         )         )
```

Figure 2.39 Fiveand.mac

of a five input AND gate in the Weinberger array without CONDs, Figure 2.43, the resulting Weinberger array logic generated by MacPitts, and Figure 2.44, the logic gate

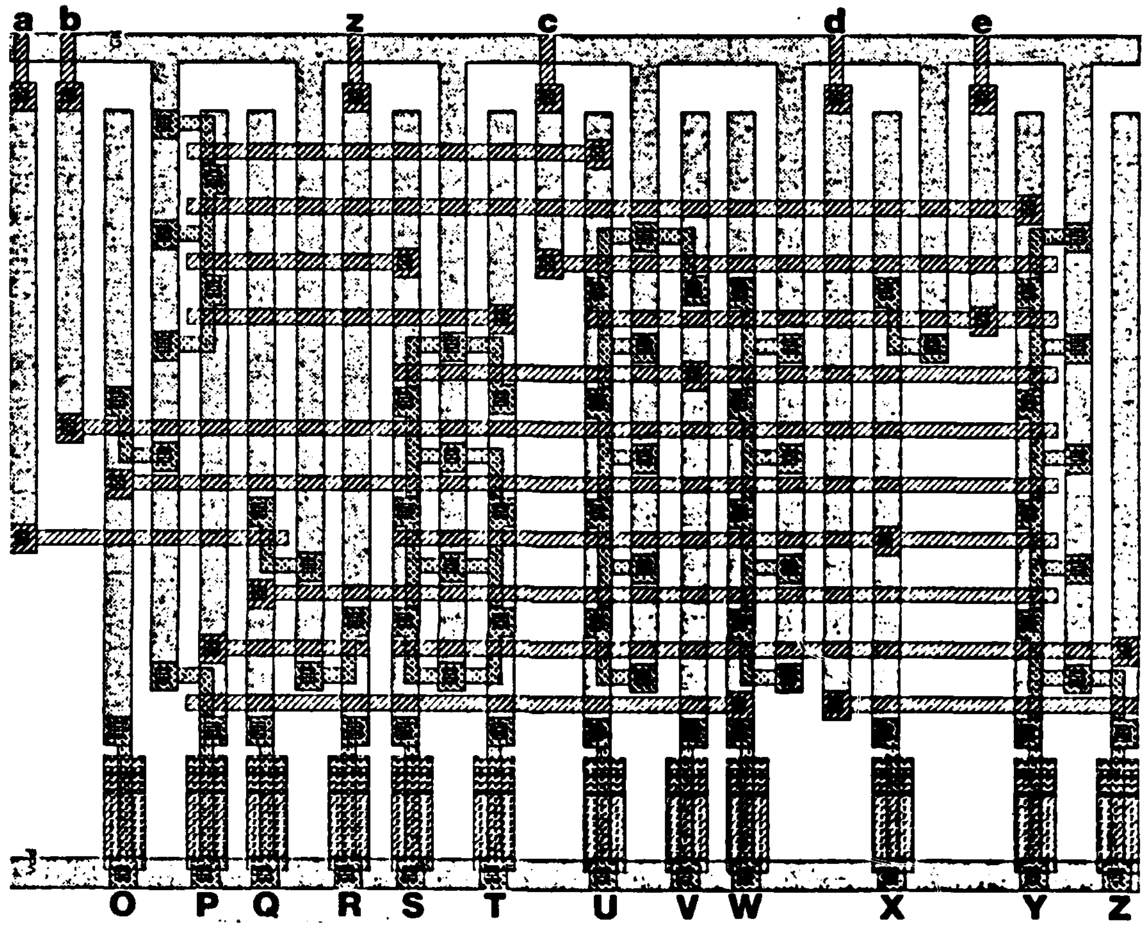


Figure 2.40 Weinberger Array from Fiveand.cif

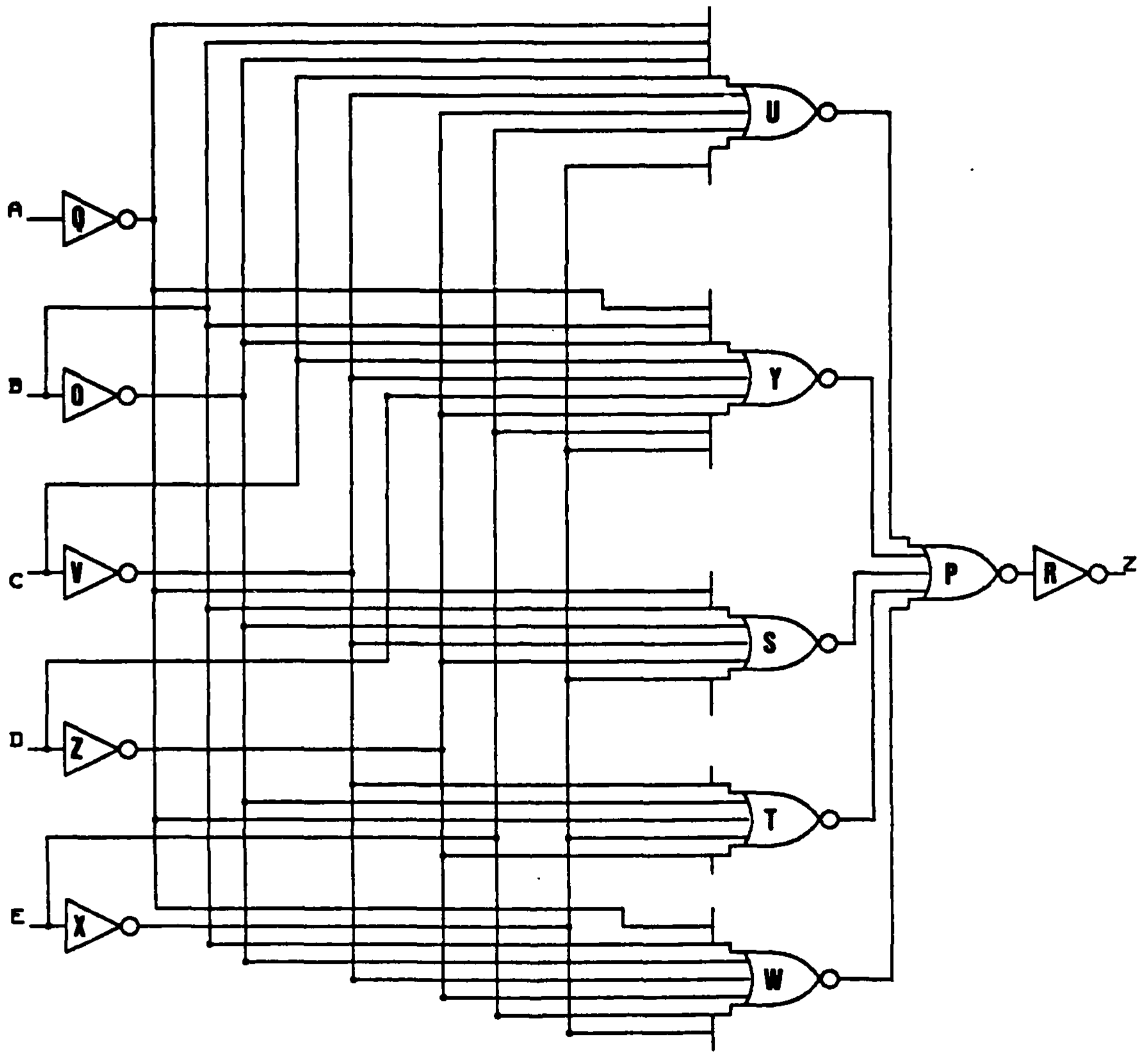


Figure 2.41 Gate Equivalent of Fiveand Logic

equivalent of a five input AND gate without CONDs. The second structure is far simpler topologically, having only six pullup transistors. The Weinberger array which achieves the same results with CONDs, Figure 2.39, requires twelve pullups by comparison. Since fewer explicit decisions need to be specified, even the code of the COND-less chip is more terse than its COND decision counterpart. In comparing the logic gate circuit equivalents, the five input AND gate created with CONDs requires six inverters and six NOR gates, and the NOR gates have fan-ins of five, six, seven, eight, and nine. There are four levels to this structure. The five input AND gate created without CONDS has only five inverters and one NOR gate with a fan-in of five, and there are two levels of gates. The circuit created without CONDs is smaller, simpler, and faster.

```
;SIMPL5AND.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<and>  FUNCTION BY MACPITTS SILICON COMPILER//5  input gate//
(program simpl5and 1
        (def 1 ground)

        (def a signal input 5)
        (def b signal input 6)
        (def c signal input 7)
        (def d signal input 8)
        (def e signal input 9)
        (def z signal output 1Ø)

        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 11 power)

        (always
                (setq  z (and a b c d e)  ) )  )
```

Figure 2.42 Simpl5and.mac

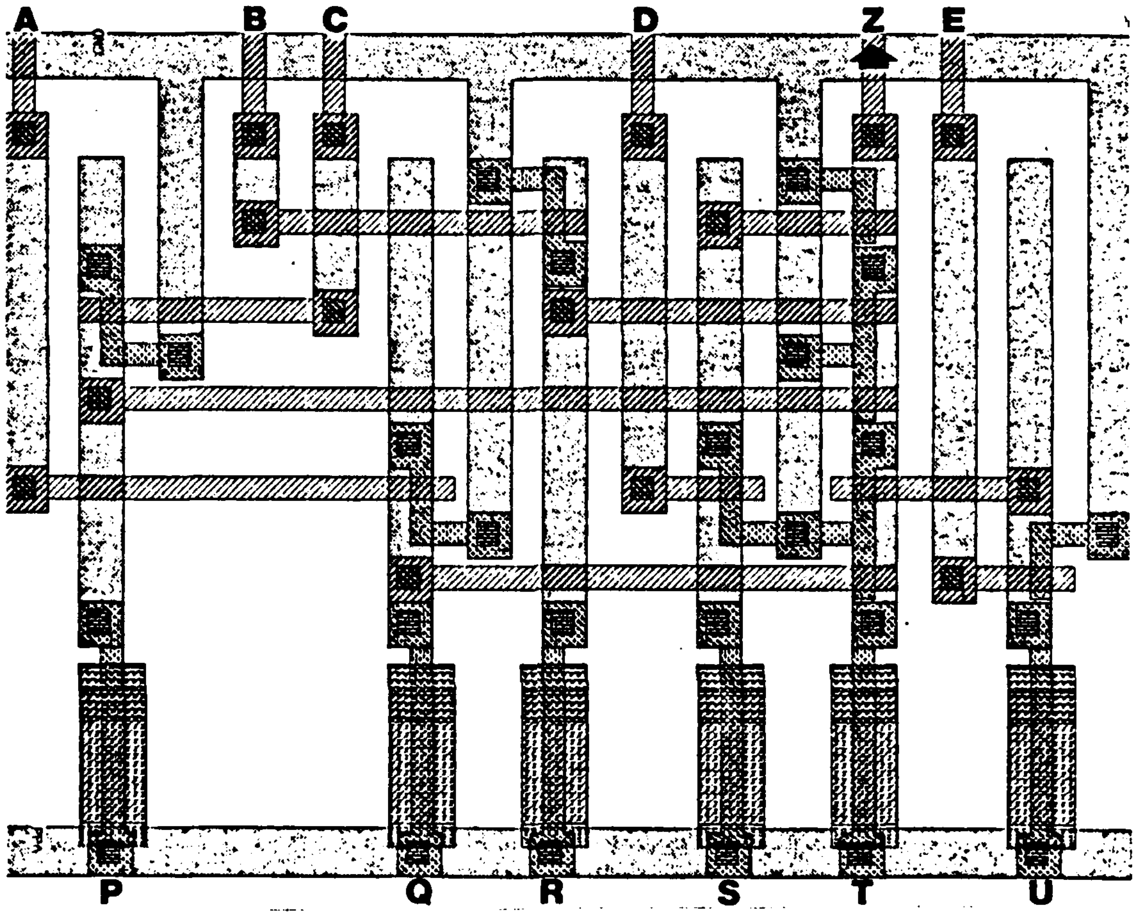


Figure 2.43 Weinberger Array from Simpl5and.cif

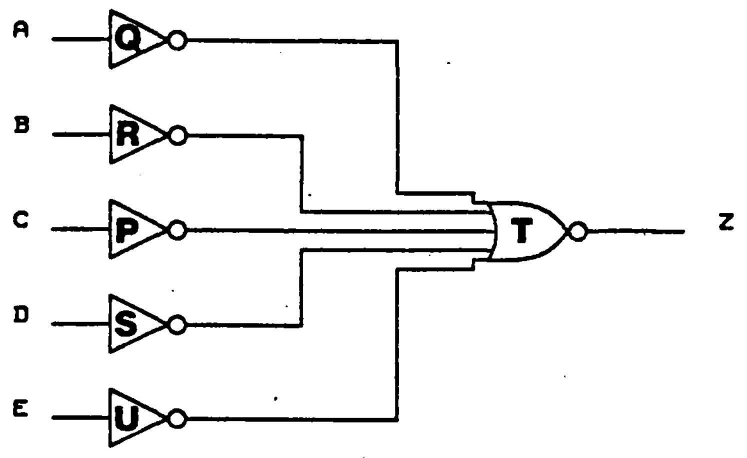


Figure 2.44 Gate Equivalent of Simpl5and Logic

The economics of using CONDless algorithms does not always justify their use. Silicon compilation is intended to free the engineer from the micro-design aspects of creating a chip, and Boolean minimization (see the home temperature controller example) is a step away from this goal. Typically, the control path is not used to implement combinational logic functions, but rather to provide controlling inputs to data path operations. The decision to signal on five simultaneous TRUE inputs would always be done as shown in Figure 2.42, and not as in Figure 2.39, but this decision would usually have a COND embracing (around) itself. The COND in MacPitts is used for decision. Attempts to minimize CONDs will lead to a loss of clarity in the algorithm (see the simplified home temperature controller example). Nevertheless, if the Weinberger array becomes too large and slow, Boolean reduction techniques such as Quine-McCluskey or Karnaugh maps should be considered.

9. A Better 15 Input Control Path OR Gate

A remarkable power savings in the Weinberger array can be expected where this alternate algorithm (explicit specification of outputs without use of COND testing) is feasible. Figure 2.45 depicts another method of algorithmically specifying a sixteen input logical OR selector in the control path (compare with Figure 2.37) . Figure 2.38 shows the resulting layout from the algorithm using multiple CONDs for selection, and Figure 2.46 shows the Weinberger array layout resulting from the algorithm using just Boolean logic specification. Figure 2.47 shows the logic gate equivalent of Figure 2.46.

```
;SMPLMLTR.MAC
;SOURCE CODE FOR ALGORITHMIC CREATION OF LOGICAL
;<or>  FUNCTION BY MACPITTS SILICON COMPILER//16  input gate//
;a simplified structure resulting from elimination of "cond"
(program smplmltr 1
          (def 1 ground)

          (def a signal input 5)
          (def b signal input 6)
          (def c signal input 7)
          (def d signal input 8)
          (def e signal input 9)
          (def f signal input 10)
          (def g signal input 11)
          (def h signal input 12)
          (def i signal input 13)
          (def j signal input 14)
          (def k signal input 15)
          (def l signal input 16)
          (def m signal input 17)
          (def n signal input 18)
          (def o signal input 19)
          (def p signal input 20)
          (def q signal output 21)

          (def 2 phia)
          (def 3 phib)
          (def 4 phic)
          (def 22 power)
          (always
          (setq q (or a b c d e f g h i j k l m n o p) ) ) )
```

Figure 2.45 Smplmltr.mac

Note in particular the difference in number of pullup transistors between the two circuits (Figures 2.38 and 2.46). There are thirty pullups in the circuit created using COND testing, and only two pullups in the circuit created from the COND-less algorithm. The pullup transistors are always turned on, and as a consequence consume proportionally more power than transistors which are intermittently turned on. So a circuit power consumption

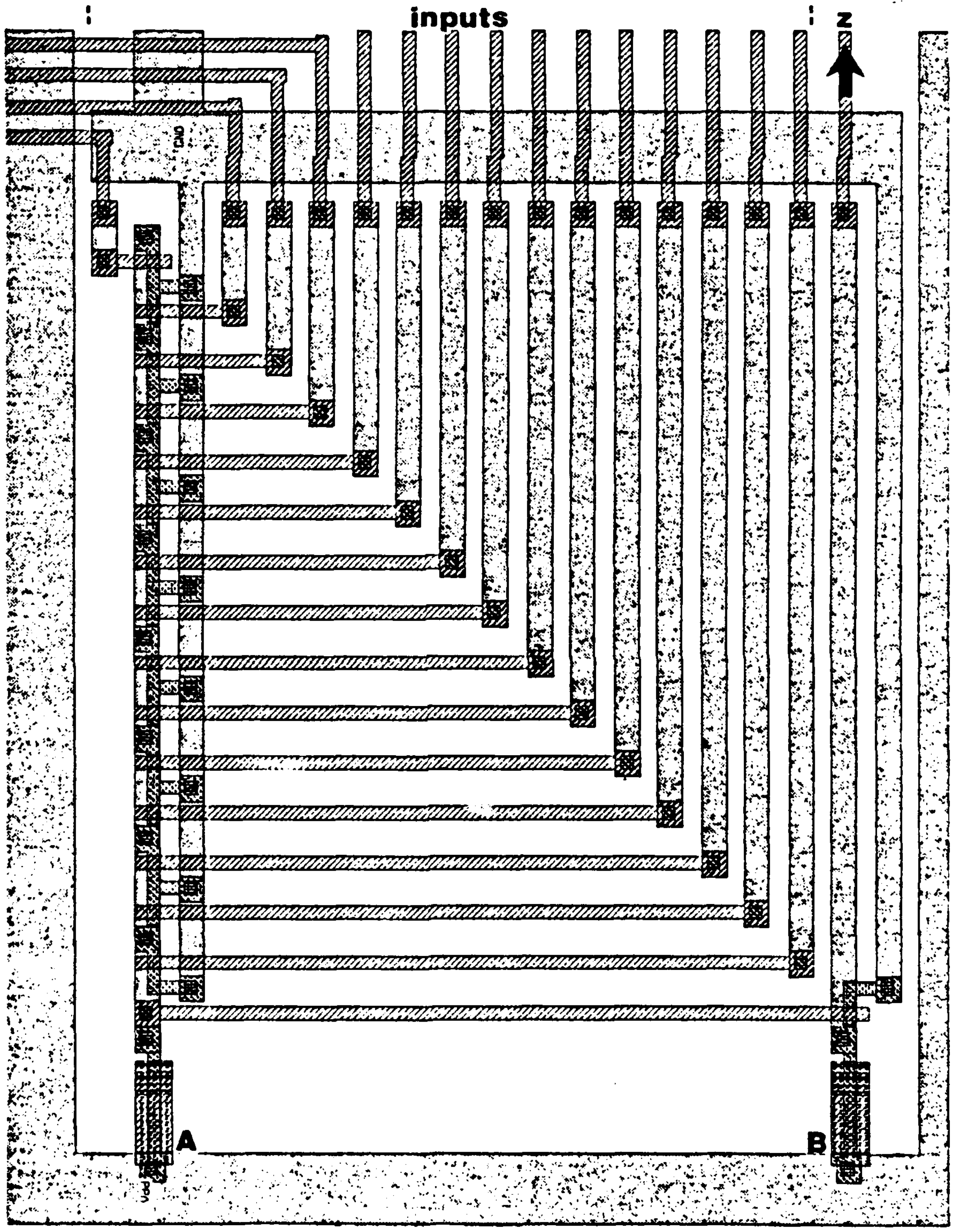


Figure 2.46 Weinberger Array from Smplmltr.cif

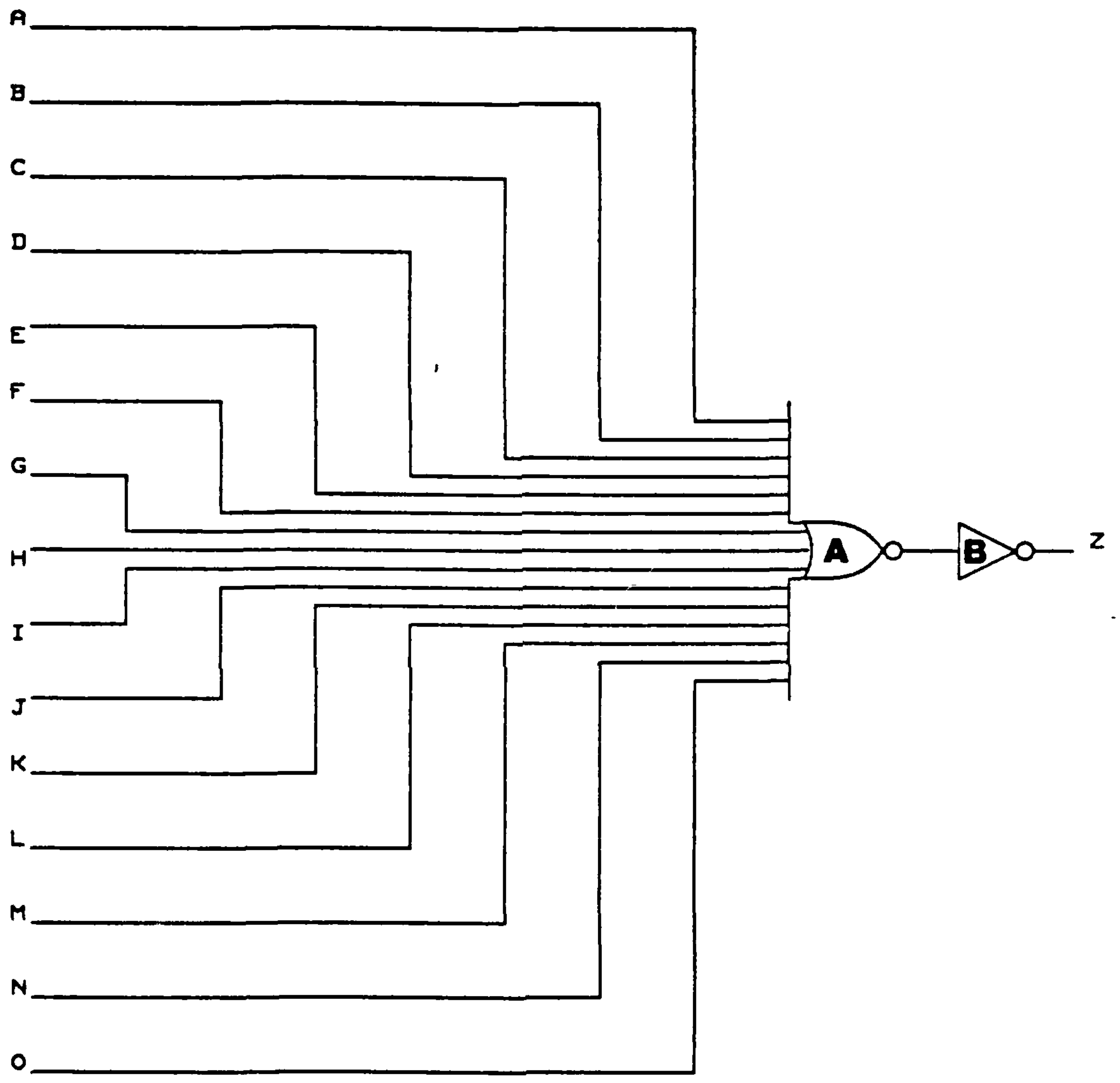


Figure 2.47 Gate Equivalent of Smplmltr Logic

savings can be realized by appropriate COND-less decision specification, where appropriate. But note that this is not always possible, nor is the COND-less algorithm always as clearly understood as the algorithm using COND for testing and branching.

These logic decisions would all occur electrically in the Weinberger array (equivalently; occurring algorithmically in the compiled LISP object code), since the decision stipulations are Boolean and not integer. The forms for Boolean combinational logic and integer (word) combinational logic are syntactically different, and it is necessary that the MacPitts programmer understand this syntax difference in addition to the logical implementation difference described previously.

10. Two Considerations In MacPitts Programming

MacPitts is both a programming language and a method of designing digital circuits. As such, the programmer must consider the consequences of syntax used in the driver algorithm (the .mac file). It is not always apparent beforehand whether a given function should be done in the control path or in the data path. The choice is determined by the syntax used by the designer.

Suppose a four input AND gate is to be designed in both the data path (word type) and in the control path (Boolean type), where a, b, c, and d are inputs and z is the output. The statement which relegates the decision to the data path is

```
  (setq z (word-and a (word-and b (word-and c d)))   )
```

where a, b, c, d, and z must all be either ports or registers (integer valued). The corresponding statement for the control path is

```
  (setq z (and a b c d))
```

which requires that a, b, c, d, and z all be either signals or flags (Boolean valued).

In complicated architectures and most sequential machines, this choice does not have to be made a priori, but rather will be made by syntax in writing the MacPitts algorithm. In simpler architectures, like a Hamming error detector or a Grey code decoder, this decision should be made beforehand. The choice can be regarded as one between individual treatment of the data bits (usually done in the control path logic), or treating the data as n-bit words (done exclusively in the data path). Examples of algorithms to do Grey code decoding and Hamming error detection and correction are given in Chapters IV and VI.

The MacPitts programmer/designer must also consider the hardware ramifications of syntax. The algorithm chosen to implement a function in MacPitts drives the circuit implementation to achieve that function.

It has been mentioned previously that COND forces conditionals to be tested in parallel, and their antecedent actions to be SETQ'd in parallel. This equates to silicon area/speed tradeoff on the chip. If multiple operations of the same type are to be done under a COND, MacPitts will instantiate copies of the required organelle, and perform the operations in parallel. Conversely, if the same operations are not put under a COND, MacPitts will instantiate only one copy of the organelle, and perform the operations serially. For instance, there are two ways to perform a set of three data path logical two-bit ANDs on six inputs. The first method does the operations in parallel, at the cost of silicon area.

```
(cond (t
          (setq x (word-and a b))
          (setq y (word-and c d))
          (setq z (word-and e f)) )  )
```

This algorithm fragment would execute in one clock cycle, but MacPitts would implement it with three data path AND gate organelles, each gate having two inputs. The slower algorithm would be

```
(setq x (word-and a b))
(setq y (word-and c d))
(setq z (word-and e f))
```

The second example would require three clock cycles to execute, but only one data path AND organelle would be instantiated. Similarly, PAR forces all forms within its scope to be executed in parallel. The best way to verify this is to create a short FSM algorithm, and manually clock it while in the interpreter. (This is also an excellent method to optimize algorithms for throughput by paralleling operations where possible and testing for execution in the interpreter. The results may not be what is expected.)

C. SUMMARY

This chapter discussed the differences between MacPitts' implementation of combinational logic in the control path and data path. The fundamental difference is one of structure, which is driven by syntax.

When the data type is defined Boolean, and the correct operations are applied to the bits, the combinational logic occurs in the control path. Control path logic is always done by a Weinberger array, an array of NOR gates. When the data type is defined as integer, and the correct operations are applied to the words, the combinational logic occurs in the data path. The fundamental units of the data path are two-input organelles, which are structural mappings of the syntactical statements NOT, AND, NAND, OR, NOR, XOR, increment/decrement, and add/subtract. The data path performs the arithmetic functions and also generates signals to control for decisions. Combinational logic syntax (and hence structure) in the data path obeys the fundamental laws of Boolean algebra, such as associativity and commutativity. The designer must consider these laws in writing the MacPitts algorithm if correct function is desired.

The LISP-like COND form produces parallelism in MacPitts. The COND form is a statement which (structurally) implements decisions in the Weinberger array and (algorithmically) drives control flow in both the .mac file and the .obj file. Control path structures may be reduced in size (where possible) by not using the COND form to specify output conditional setting. The alternative is the PAR (parallelize) form, which parallels all the forms under its scope. The forms embraced by PAR must be the functional equivalents of those under COND, which requires designer intervention and possibly Boolean algebraic reduction. The result of this alternative is unconditional explicit assignment of outputs. This is feasible in simpler chips, and should always be considered on the basis of an engineering tradeoff between design time and chip speed.

The COND statement, with multiple selections of conditionals, can be viewed as an implicit AND-OR structure realized in NORS in the Weinberger array. An alternate syntactical viewpoint of COND is the CASE statement.

The gates created in this chapter are rather artificial, in that they were made to show just the structures desired. In practice, the combinational logic structures used are likely to differ slightly.

III. A SPEED-POWER COMPARISON BETWEEN A DATA PATH AND CONTROL PATH EQUIVALENT CIRCUIT

A behavior-oriented silicon compiler requires a high level algorithmic description of the chip's desired function as its input. The output is a machine readable low level geometric description of the resulting digital circuit, usually CIF (Caltech Interchangeable Format), a language describing rectangles from which the various process masks and their relative locations are registered. When a CIF file is processed by Mosis (Metal Oxide Silicon Implementation Service), the desired chip results.

Chapter II considered the qualitative effects of algorithmic syntax on some circuit structures in the data and control paths. It is also desired to do a quantitative investigation on functionally equivalent circuits in each path, and to compare the results. The circuits chosen are the five input AND gates in both their control path and data path configurations. Handcrafted versions of the five input AND gate are contrasted to the MacPitts five input AND gates.

A. DATA PATH FIVE INPUT AND GATE

Figure 3.1 shows the algorithm used to create a five input AND gate in the data path. Figure 3.2 shows the labelled cifplot of the four cascaded NAND organelles and four inverters, and Figure 3.3 is the logic gate equivalent of the cifplot. The LISP object file is included in Appendix A to show how MacPitts implements the data path AND function

```
;FIVAND.MAC. data path
(program fivand 1
        (def 1 ground)
        (def a port input (2))
        (def b port input (3))
        (def c port input (4))
        (def d port input (5))
        (def e port input (6))
        (def z port output (7))
        (def 8 phia)
        (def 9 phib)
        (def 10 phic)
        (def 11 power)
        (always
        (setq z
(word-and a(word-and b(word-and c(word-and d e))))))
```

Figure 3.1 Data Path Five Input AND Gate .mac File

by invoking the organelle AND four times. As discussed in Chapter II, the MacPitts algorithm produces the LISP object file, from which MacPitts (the silicon compiler) produces the layout. At run time, the MacPitts (silicon compiler) script file shown in Appendix A is created. The best way to create a script file of a MacPitts terminal session is to issue the command

```
macpitts basename herald > basename.script &
```

where the option herald directs MacPitts to send compiler messages (see compmesg.* files in MacPitts source code) to the designated output device, ">" is the BSD Unix redirect,

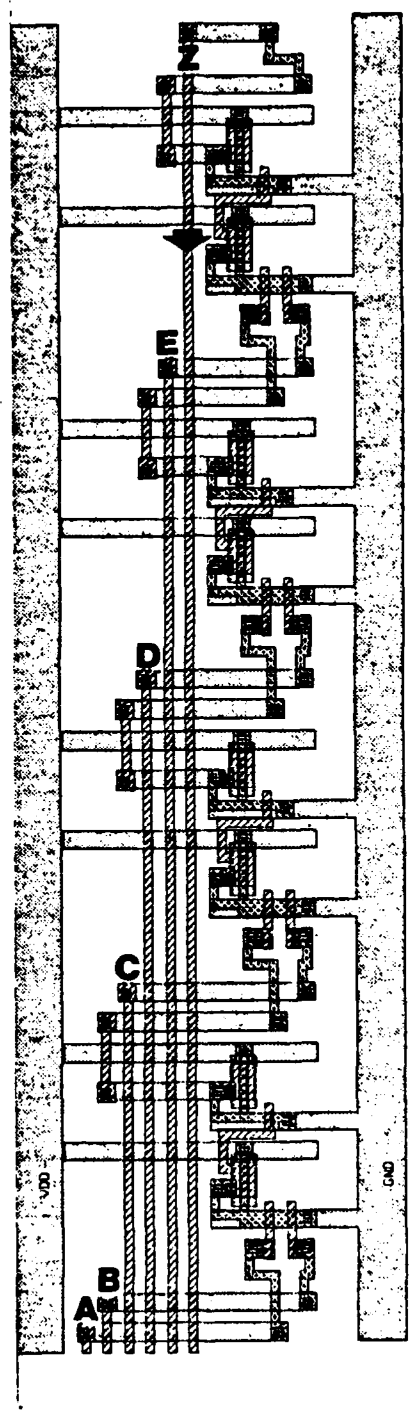


Figure 3.2 Stipple Plot of Data Path Five Input AND Gate

basename.script is the file into which the terminal session is to be recorded, and "&" is the Unix command to put a process into the background. If the algorithm is not fully debugged, then issue instead

```
macpitts basename herald
```

so MacPitts diagnostics and Liszt diagnostics both will come to the screen, and no hardcopy recording will occur. It is possible to both monitor and simultaneously record the MacPitts compilation, by issuing the command

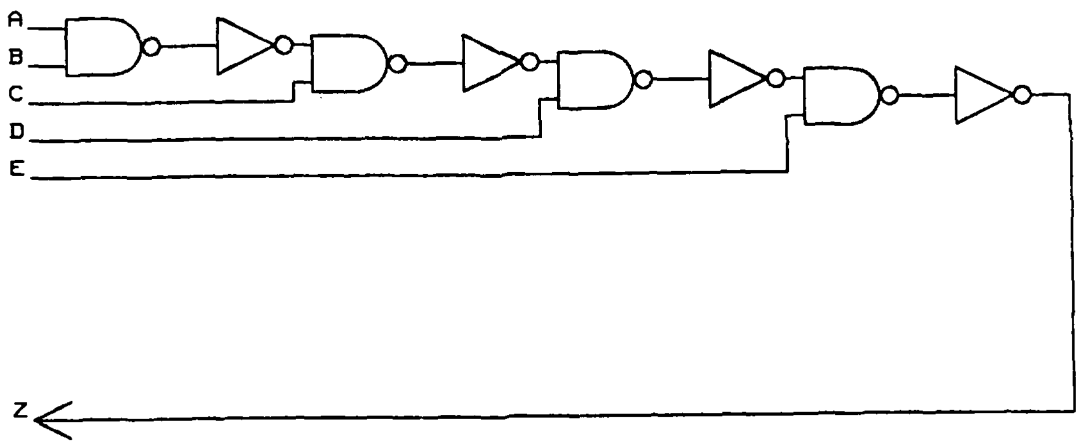


Figure 3.3 Gate Equivalent of Figure 3.2

```
script basename.script   (starts script recording)
```

to which Unix will respond with

```
"script started, filename is basename.script"
```

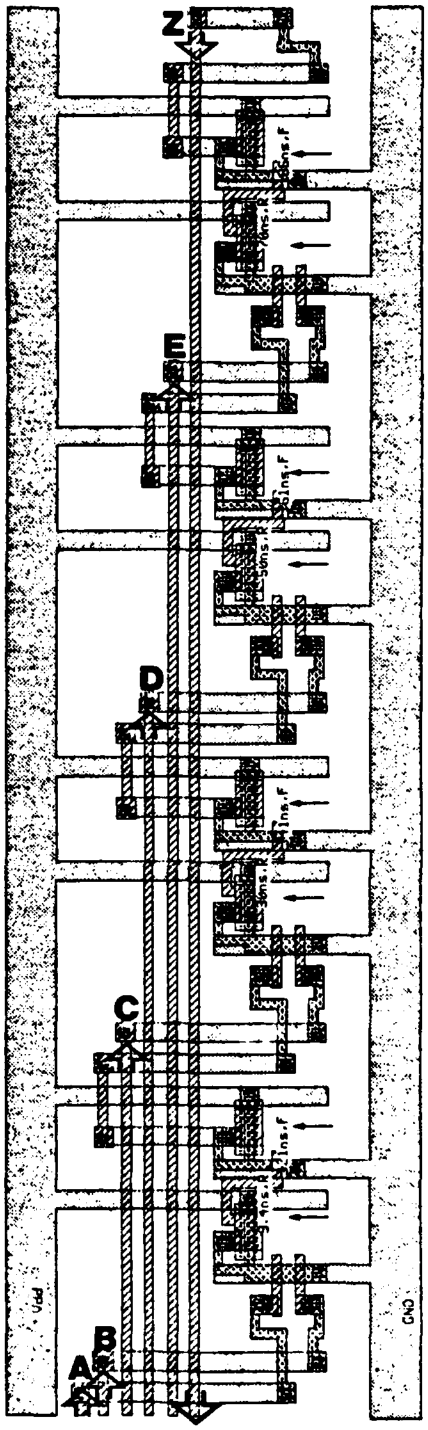


Figure 3.4 Stipple Plot Showing Critical Nodes

then issue the full path command (a Unix bug requires this)

```
/vlsi/macpit/bin/macpitts basename herald
```

and when compilation is done type control d to terminate the script recording. The script capability is useful for following the MacPitts compilation process, gives insight into how MacPitts works, and assists in debugging the driver algorithm. Tracing of MacPitts' compilation of an algorithm can then be done with a grep search on the compmesg.* files for the statistics and the h1.lisp files for the herald messages. If the algorithm halts execution, the script file indicates where in the compilation process the error was detected. That part of the algorithm can then be checked for errors.

The script of a MacPitts session also has informative material (statistics) on the chip size, components, maximum power used, and host computer effort expended to compile the chip. Carlson [Ref. 2:p.43] describes the script file produced by a MacPitts compilation session.

After the basename.cif file is produced by MacPitts, it is necessary to comment out the beginning user extension zero lines with the vi screen editor. This is done by invoking vi on the cif file

```
vi basename.cif
```

and placing parentheses around these lines. Carlson

[Ref. 2: p.70] explains why this is necessary. The Caesar file must next be created so labelling of nodes can be done for Mextra (Manhattan Circuit Extractor). The command to convert a .cif file to a .ca file is

```
  cif2ca -o <offset> basename.cif
```

where the offset is a number added to the Caesar symbolxx.ca files to distinguish them from previously created symbol files which might have the same number (xx).

The procedure described above results in a MacPitts end product, the basename.cif file, and a version of that file amenable to editing in the VLSI graphics editor Caesar, the basename.ca file. For quantitative analysis of a MacPitts design, further steps are required.

To begin this analysis, the nodes are labelled (in Caesar) for Mextra and Crystal (a timing analyzer). Work by Froede [Ref. 3:pp 63-80] addresses Crystal analysis of MacPitts circuits. After the input, output, GND, and Vdd nodes are labelled, the following commands are issued

```
  :save
```

and then,

```
  :cif -p
```

in Caesar to save the new labelled .ca file and to create a .cif file with nodes at points (-p) for Mextra. Figure 3.2 is the point-labelled cifplot of the data path five input

AND gate. Next Mextra is invoked on the labelled file by the command

```
  mextra -o basename
```

where the -o switch causes more accurate capacitance calculation (than is done without -o). Mextra produces the basename.nodes file, which can be checked for connectivity and to see that all labelled nodes are included. Appendix A shows the .nodes file for the data path AND gate. The basename.sim file is also produced, and can be used for switch level simulation with Esim, SPICE simulation, Crystal timing analysis, and power estimation with Powest. The berk85 version of Crystal is the more useful (compared to the berk83 Crystal) version. To record a Crystal session, start the script recording, and then call Crystal with its full path designator

```
  /vlsi/berk85/bin/crystal  basename.sim
```

Crystal has many options and commands. The 1985 version of the Crystal manual which describes them is available on the Naval Postgraduate School VAX 11/780 in the file

```
  /vlsi/berk85/doc/crystal/crystal.tblms
```

Appendix A shows the script recording of a Crystal analysis of the data path AND gate. After the input and output nodes are assigned and the delay is given, the command

```
  critical  -g  filename.dummy
```

is issued, then Crystal is stopped with

```
  quit
```

and then script is terminated with control d. The critical command determines the time-critical (i.e., slowest) signal path, and the -g (graphical results) switch in conjunction with it creates a Caesar-compatible file of the critical node locations as shown in Appendix A. This file can then be added to the basename.ca file by the sequence of commands

```
  caesar basename        (Caesar edit labelled file)
  :source filename       (add critical nodes to screen)
```

Since the Crystal nodes displayed in Caesar are not reproduced in cif, the nodes must be edited in Caesar if an annotated stipple plot is desired. One technique is to erase the Crystal-sourced (created by the :source command) nodes, and replace them with implant layer squares (implant for visibility and contrast) and then to relabel the delay times with Caesar's :label command. The revised Caesar file can then be saved and converted to cif for stipple plotting with the series of commands

```
  :save
```

and then

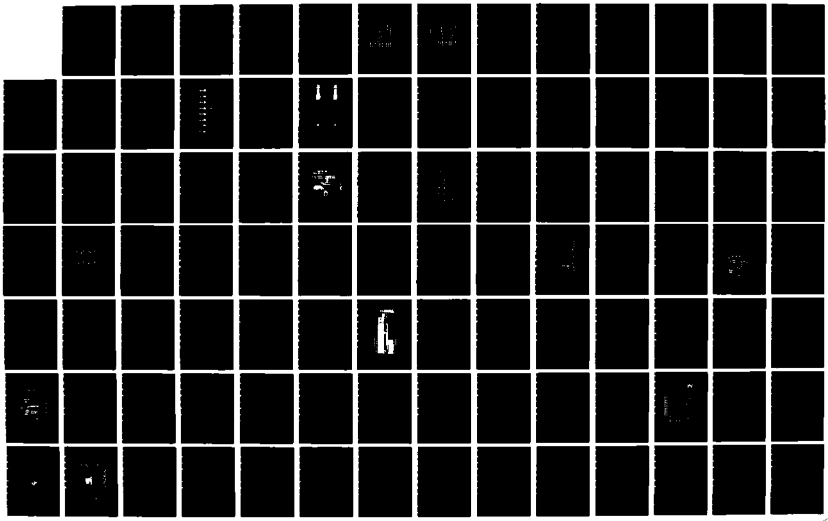
```
  :cif -p
```

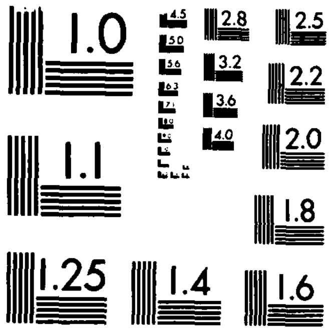
Figure 3.4 shows the cifplot of the circuit with the critical nodes marked. The critical nodes lie along what Crystal considers the critical (slowest) path. The largest delay shown is the circuit cumulative delay, and each marked node indicates a cumulative delay. This makes it simple to determine the delay between critical nodes as the difference between their successive cumulative delays. The stipple plot can be difficult to interpret if it is desired to determine what structure causes the delays. A gate equivalent of the cifplot can be helpful in the analysis. The gate level equivalent of this circuit with marked cumulative delays is shown in Figure 3.5. The data path AND gate spreads the delay out evenly, with approximately 10 ns per gate, as is expected from the transistor aspect ratios shown in Figure 3.2.

The maximum power consumed by the circuit can be determined in either of two ways. The MacPitts script session (of the compilation process) records it, or Powest (Power ESTimator) can be used on the basename.sim file produced by Mextra. Powest computes the power based on only the number of depletion transistors, assuming that they are on all the time (for the maximuum power figure) or on half the time (for the average power figure). MacPitts considers both the number of depletion transistors and the power consumed by the circuit wires, so the MacPitts power should be the more accurate of the two. The command to use Powest

AD-A161 098 VLSI (VERY LARGE SCALE INTEGRATED CIRCUITS) DESIGN WITH THE MACPITTS SILICON COMPILER(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R C LARRABEE SEP 85 2/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

on the .sim file is

```
  powest -p < basename.sim
```

Where the -p switch directs Powest to print out informative data about the circuit, and the < is the Unix backwards redirect, which directs the .sim file to Powest. Appendix A shows the result of a Powest analysis of the five input data path AND gate. Checking the Powest result can also serve as a check on the accuracy of Mextra's nodal extraction. For example, from Figure 3.2, the cifplot, there are eight depletion pullup transistors and no enhancement pullups or special transistors. The Powest analysis in Appendix A confirms this count. This transistor count verification is important in a MacPitts data path design analysis. It has been observed that the Vdd bus (top metal trace, Figure 3.2) does not always connect with the vertical lines to the pullup transistors. The gap is so small that it is not usually evident in Caesar, although a design rule checker such as Lyra will detect it.

B. CONTROL PATH FIVE INPUT AND GATE

Chapter II discussed the two different types of control path five input AND gates possible. The COND driven AND gate was structurally more complicated (Figure 2.40), while the "CONDless" AND gate was comparatively simple (Figure 2.43). The COND driven AND gate is more likely to occur in practice (since the purpose of the Weinberger array is decision making, or conditional control), so that circuit is analyzed in this section.

Figure 3.6 is the MacPitts driver which creates the control path to implement this logic. Figure 3.7 is the resulting Weinberger array, which has had the odd_partial_gate input gap errors repaired in Caesar (so Lyra and hence Mextra will work, and produce a valid .sim file). Figure 2.41 is the logic gate equivalent of the Weinberger array. Appendix A contains the object file for this chip. The NOR character of the Weinberger array logic was discussed in Chapter II, and in the LISP object file all logic is done with NORs. Appendix A also contains the LISP object file for the equivalent data path function, and in Figure 3.2 all logic is implemented in AND organelles. The Weinberger array is composed of inverters also, but an NMOS technology inverter is just a degenerate (single input) NOR gate. The difference in implementation from a software (language) perspective is that the data path function is done in organelles, and the control path function is done exclusively in NORs. The data path organelles are already compiled in the organelles.lisp files, so MacPitts has to work harder to create the equivalent function in the control path. Both the basename.obj file and the cifplot of the Weinberger array show the NOR logic implicit in control path combinational logic. The MacPitts script file is shown in Appendix A, and its data path counterpart is also for

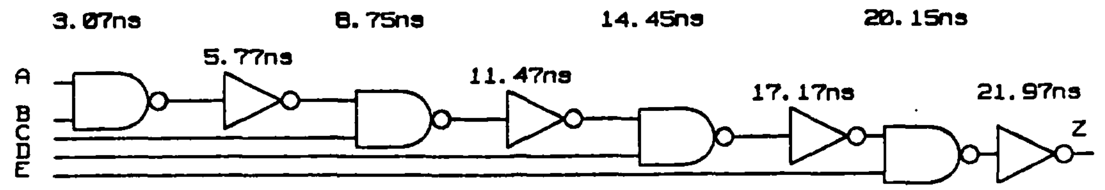


Figure 3.5 Gate Equivalent of D.P. AND Showing Delays

```
;FIVEAND.MAC, control path gate
(program fiveand 1
        (def 1 ground)
        (def a signal input 5)
        (def b signal input 6)
        (def c signal input 7)
        (def d signal input 8)
        (def e signal input 9)
        (def z signal output 1Ø)
        (def 2 phia)
        (def 3 phib)
        (def 4 phic)
        (def 11 power)
        (always
        (cond  (a
                 (setq  z (and a b c d e)  ) )
               (b
                 (setq  z (and a b c d e)  ) )
               (c
                 (setq  z (and a b c d e)  ) )
               (d
                 (setq  z (and a b c d e)  ) )
               (e
                 (setq  z (and a b c d e)  ) )
               (t
                 (setq  z   f              ) )
                              )     )       )        )
```

Figure 3.6 Control Path Five Input AND Gate .mac File

comparison. These files contain information which will be compared in the next section.

The same CAD tools were used on this circuit as were used on the data path circuit, in the same order. Mextra produces the .nodes file (Appendix A). The control path logic also differs from the data path logic in the number of nodes produced to model the equivalent circuit. The Weinberger array node list is approximately 25% larger than the equivalent data path node list. Appendix A contains the Crystal analysis of the circuit, and the critical path file for source input to the Caesar file. Figure 3.8 depicts the Weinberger array with the critical nodes marked, and Figure 3.9 is the gate level equivalent of Figure 3.8 with delay node values and gate equivalent fan-ins marked. Appendix A contains the Powest analysis of the control path AND gate, and this information is incorporated into the following table for comparison.

C. SPEED-POWER COMPARISON

Table 3.1 compares functionally equivalent MacPitts five input AND gates in both their control and data path configurations.

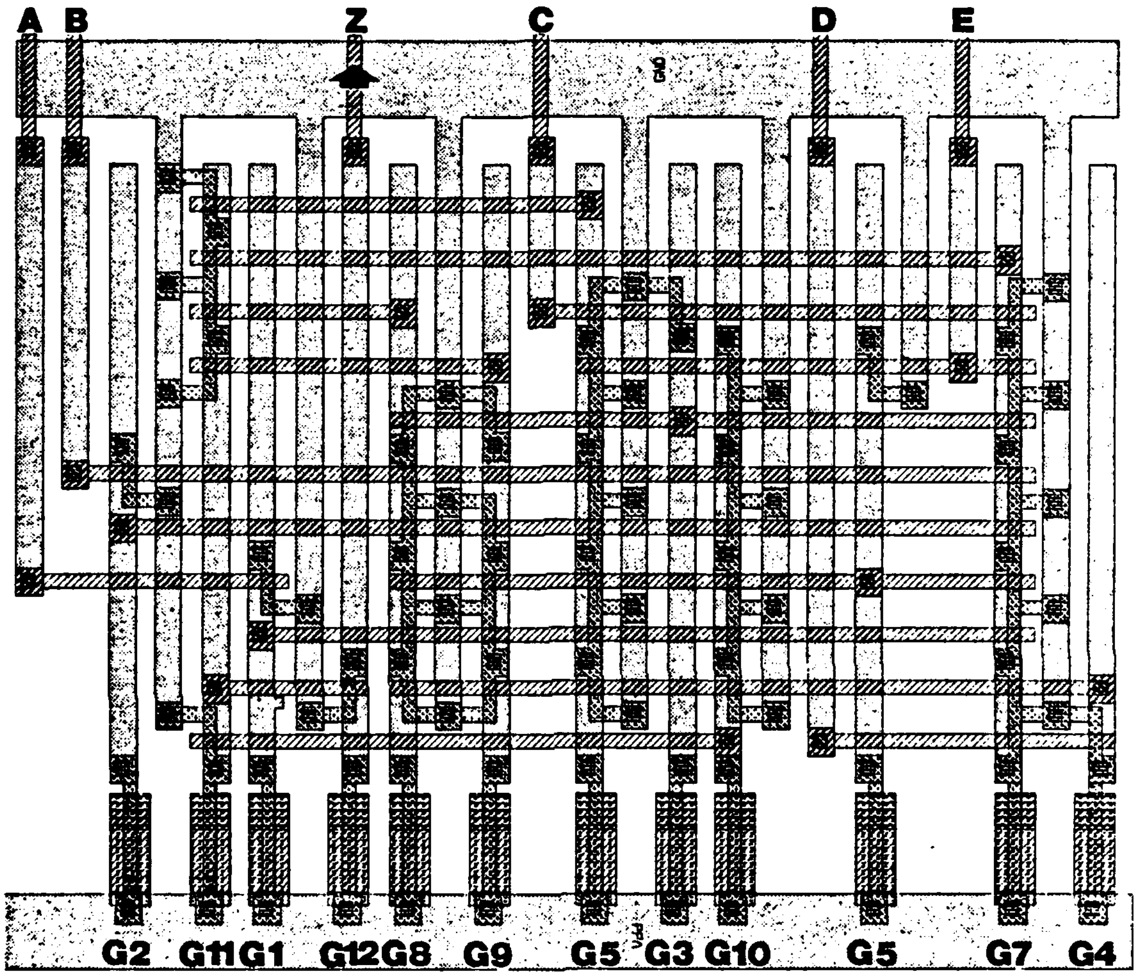


Figure 3.7 Weinberger Array From C.P. Five Input AND Gate

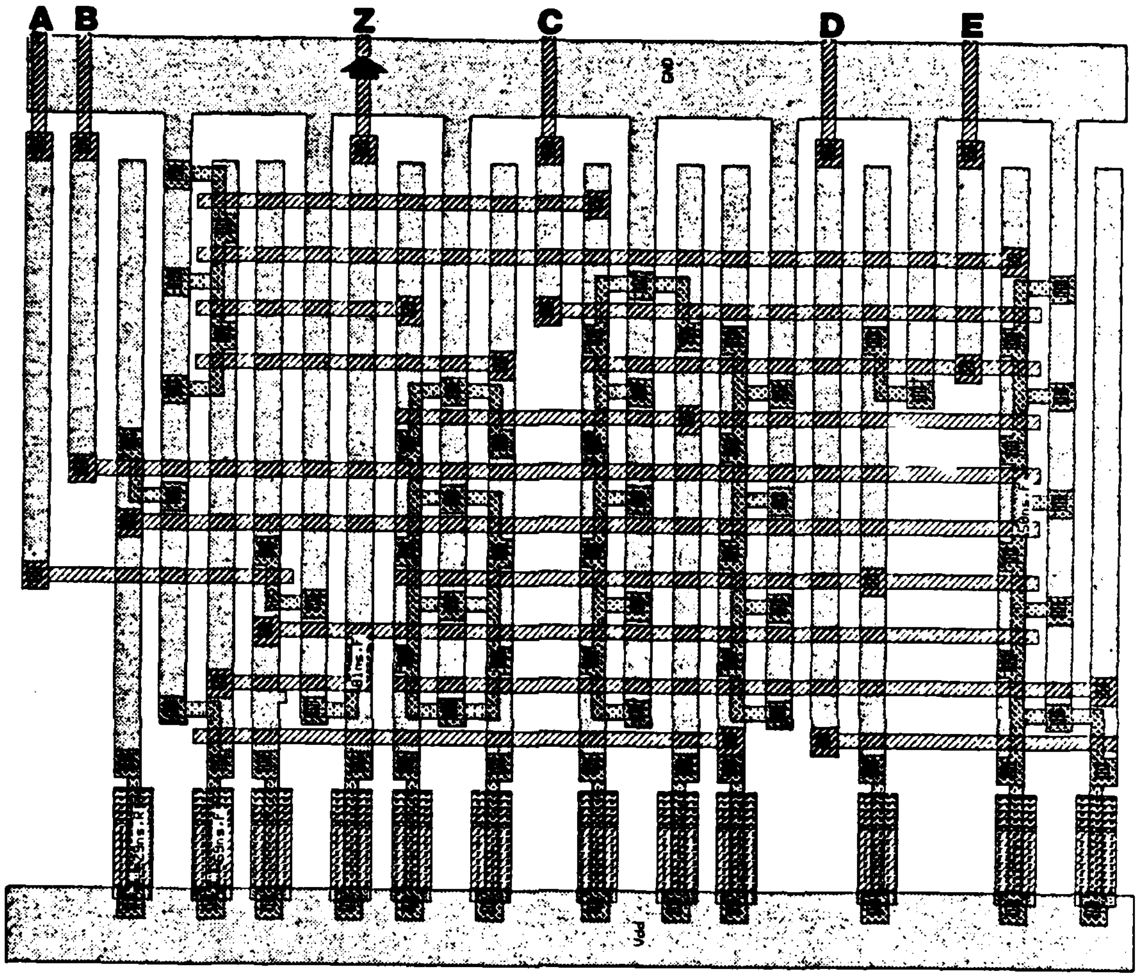


Figure 3.8 Weinberger Array With Critical Nodes Marked

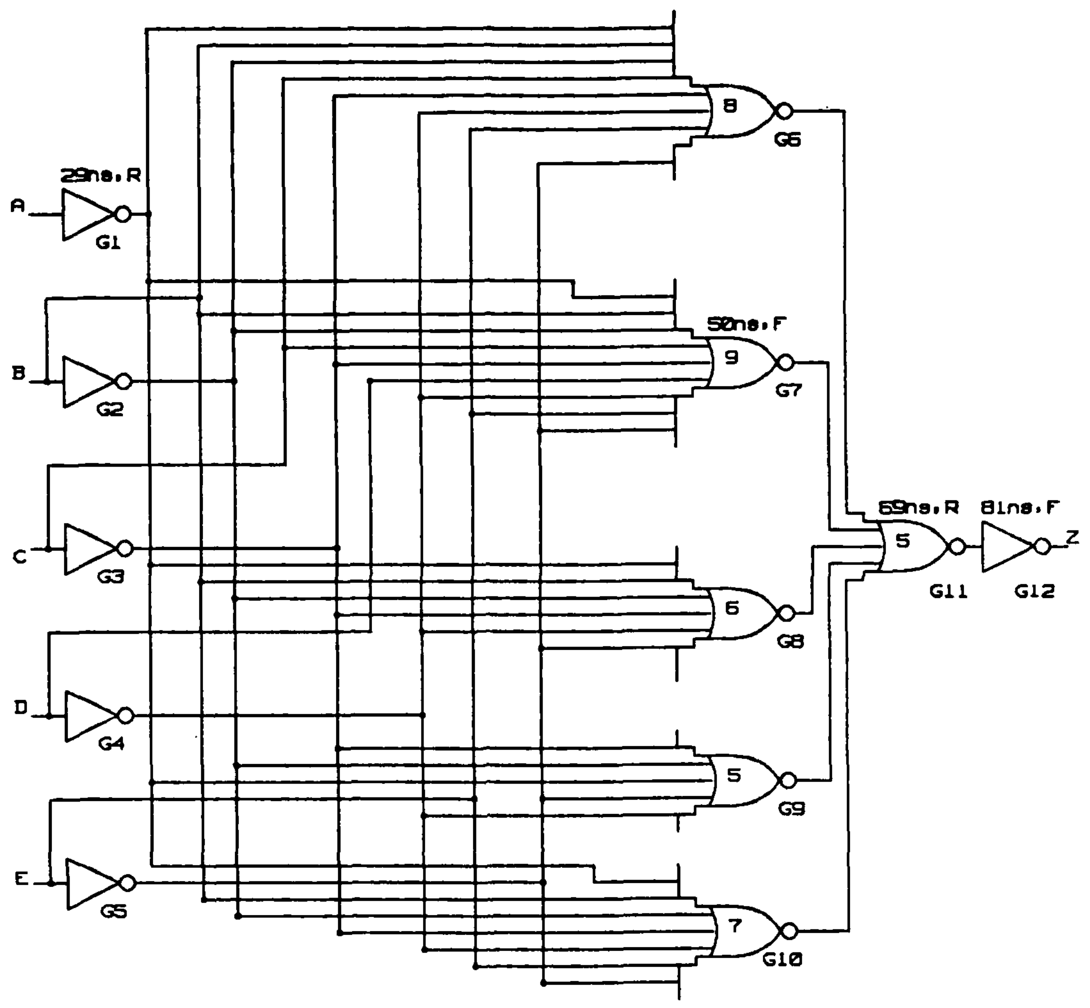


Figure 3.9 Gate Equivalent of Weinberger Array With Delays

TABLE 3.1

FIVE INPUT AND GATE

	DATA PATH	CONTROL PATH
MacPitts power [W]	.0407	.0381
Powest power average,[W]	.00182	.00094
maximum,[W]	.00245	.00188
Maximum delay Crystal,[ns]	81.15	85.98
Length x width of logic circuit [lambda]	209 x 173	386 x 113
Number pullups (less pads)	12	8
Compile time [CPU min]	2.106	1.535
CPU peak memory demand [kb]	349	357

So all other things being equal, the data path circuit is superior to the control path circuit in terms of power consumption, size, and compile time in MacPitts, and slightly inferior in terms of maximum speed attainable.

The data path power advantage is understandable when the number of depletion pullups there is compared to the number in the control path. A power consumption ratio of 0.67 is expected, and the calculated ratio is close to that. The difference is explained by the long horizontal polysilicon runs in the Weinberger array, which have a comparatively high specific resistance (ohms/square), and therefore consume more power. The first row in the table above, MacPitts computed power, is calculated on the whole chip and not on the just logic circuitry. This value shows a similar power consumption relationship, but the poly runs connecting the Weinberger array to the rest of the circuit consume additional power (the rest of the analysis in the table above is done on just the logic circuits, and not on the whole chip).

The speed of the two circuits is approximately the same. Figures 3.4 and 3.8 show the Crystal-generated delay data on the data path and control path circuits. The results are perhaps clearer in Figures 3.5 and 3.9, the logic gate equivalents of the cifplots. In the data path (Figure 3.4), the signal experiences approximately 21 ns delay per organelle. The organelle comprises a NAND gate and an inverter (Figure 2.14). From the gate equivalent and the Crystal script (Figure 3.7), each NAND gate induces a delay of 9.4 ns, and each inverter induces a delay of 11.4 ns. The circuit shown in the gate equivalent is expected to produce a delay equal to the product of the number of organelles and the delay per organelle. The expected delay is then 4 x 20.8 = 83.3 ns. The cifplot (Figure 3.2) reveals where the added three ns delay arises. The river routing routine in MacPitts difference is explained by the long horizontal polysilicon runs in the Weinberger array, which have a comparatively high specific resistance (ohms/square), and therefore consume more power. The first row in the table above, MacPitts computed power, is calculated on the whole chip and not on the just logic circuitry. This value shows a similar power consumption relationship, but the poly runs connecting the Weinberger array to the rest of the circuit consume additional power (the rest of the analysis in the table above is done on just the logic circuits, and not on the whole chip).

The speed of the two circuits is approximately the same. Figures 3.4 and 3.8 show the Crystal-generated delay data on the data path and control path circuits. The results are perhaps clearer in Figures 3.5 and 3.9, the logic gate equivalents of the cifplots. In the data path (Figure 3.4), the signal experiences approximately 21 ns delay per organelle. The organelle comprises a NAND gate and an inverter (Figure 2.14). From the gate equivalent and the Crystal script (Figure 3.7), each NAND gate induces a delay of 9.4 ns, and each inverter induces a delay of 11.4 ns. The circuit shown in the gate equivalent is expected to produce a delay equal to the product of the number of organelles and the delay per organelle. The expected delay is then 4 x 20.8 = 83.3 ns. The cifplot (Figure 3.2) reveals where the added three ns delay arises. The river routing routine in MacPitts runs the input and output lines in polysilicon, and in this case the output comes from across the circuit. The specific resistance and capacitance of polysilicon and the poly input and output line lengths constitute this added delay. Froede [Ref. 3:pp. 72-76] has validated Crystal's timing calculations and compared them for accuracy with the theory presented in Mead and Conway [Ref. 4:pp. 3-14].

Figure 3.8 is the corresponding data path cifplot with Crystal delay annotation for the Weinberger array. The structure of the Weinberger array is, at first glance, intimidating. Two observations on function assist in understanding the structure. (1) Any GND track that connects a Vdd track with only one diffusion gate is an inverter, and (2) any GND track that connects a Vdd track with multiple diffusion gates is a multiple input NOR gate. The transverse poly runs turn on and turn off the NOR gates and inverters. This cifplot shows six inverters and six NOR gates. Furthermore, multiple input/single output Weinberger arrays appear to always exhibit the four level structure shown in Figure 3.9, a bank of inverters followed by a bank of multiple input NORS followed by a single multiple input NOR followed by an output inverter. Figure 3.9 is the gate level equivalent of the Weinberger array in Figure 3.8, with delay annotation and fan-in (shown inside the bodies). The critical path is from input A to the second level nine-input NOR through the output NOR through the output inverter. The

Weinberger array total delay is then 81.15 ns, not much different from the data path circuit delay. This delay calculation only considered the Weinberger array, however, and not the connections to it which MacPitts creates in polysilicon. If these additional connections were considered, the Weinberger array would certainly be slower than the equivalent structure in the data path. Figure 3.8 shows the critical path (annotated with cumulative delay times), and it is evident that the longest delay path occurs along the wires which must charge the largest capacitances. The data path block is connected to the rest of the chip with metal lines (in most cases), so this added delay from polysilicon runs would not apply to it.

The relative sizes of the data path and control path circuitry are as expected from the object code respective descriptions. The object code for the data path instantiation is approximately half the size of the code for the control path. From a theoretical viewpoint, the cascaded AND organelle circuit is more conservative of both silicon and power than is the Weinberger array. This principle applies to most combinational logic in MacPitts, since the Weinberger array builds functions only from NOR gates, whereas in the data path the choice of building blocks is larger (NAND, NOR, and inverter). The MacPitts chip size comparison is given in the table above, but the circuit dimensions are more informative. The data path circuitry has an area of .090 square mm, and the Weinberger array covers .109 square mm, an area of 120 % over the data path functional equivalent.

The compile time for the control path chip is approximately 25% greater than for the data path chip. This is understandable in light of the gate instantiation process for each path. From the cifplots in Figure 3.2 (data path) and Figure 3.7 (control path), the circuits are not even remotely similar structurally. The data path circuit is made from quadruple instantiation of the MacPitts library AND organelle (see Appendix A, the object code). This organelle is accessed four times, its location calculated, and then it is instantiated. The control path Weinberger array (Figure 3.7) requires time consuming decisions and construction from more primitive units, NOR gate inputs (see the object code, Appendix A). The poly cross-runs must then be laid down. All of these processes are computationally intensive, and this is why large control-heavy Weinberger array architectures take a long time to compile. Chapter VI describes the design of a control path chip and how long it required for compilation.

D. ALTERNATE POSSIBILITIES FOR FIVE INPUT AND GATES

The five input AND gate, as implemented by MacPitts in both its data path and control path configurations, has been examined above. Each configuration can be improved in the areas of speed and circuit density. While the goal of silicon compilation is to free the designer from excessive preoccupation with detail, perhaps the combinational logic generation by MacPitts can be improved. The following section presents two hand-designed variants of the five input AND gate for comparison with the MacPitts designs.

The first design is patterned after the Mead-Conway cells as illustrated throughout [Ref. 4]. The layout is similar to that generated by MacPitts for the five input data path AND gate, a linear cascade of NANDs and inverters. Figure 3.10 shows the hand-crafted circuit. It is noticeably different from the MacPitts design in two ways. The pulldown transistors on the NAND gates are four lambda wide. This allows a shorter data path, while preserving the 4:1 aspect ratios of the transistors. Also, the characteristic MacPitts pullup diffusion "dogleg" is absent. This is accomplished by joining the pullup diffusion and polysilicon layers with an in-line buried contact. The circuit is also less wide than the MacPitts equivalent. MacPitts uses NAND organelles, and interconnects then with metal/poly/diffusion wires. This wastes a lot of space. In the hand-designed five input AND gate, the output is taken from the pullup on a polysilicon wire, and routed directly to the input of the next transistor. This saves (at a minimum) two contact cuts in the transistor interconnections. As expected, this configuration is also considerably faster than the MacPitts equivalent. The MacPitts data path five input AND gate requires 86 ns for signal propagation, and the handcrafted design requires 22 ns. Figure 3.11 shows the gate equivalent of the hand design, with propagation times marked above the respective gates.

This configuration is amenable to silicon compilation if the NAND-NOT pairs as shown are incorporated into the MacPitts organelle library as an AND organelle. Similar speed and area enhancements are expected for other data path logic gates.

If the multiple input AND gate can be improved so much using the basic MacPitts data path cascading scheme, does a better method exist using another approach? The drawback to the cascading scheme is the linear pileup of transistors. This requires more silicon, and consequently more current to charge the gates of later stages. A better design would use only one gate for the five input AND function, as shown in Figure 3.12. This is a true five input AND gate, as opposed to the previous circuits which only emulate the five input AND function. The circuit is much smaller than the previous five input AND gates, and is much faster. Figure 3.13 is the gate equivalent with marked delays. This circuit is patterned after those circuits illustrated in [Ref. 4] also. The wide (10 lambda) pulldown region permits a comparatively short transistor (i. e., the pullup aspect ratio is not very large). The multiple input NAND and NOR derivatives

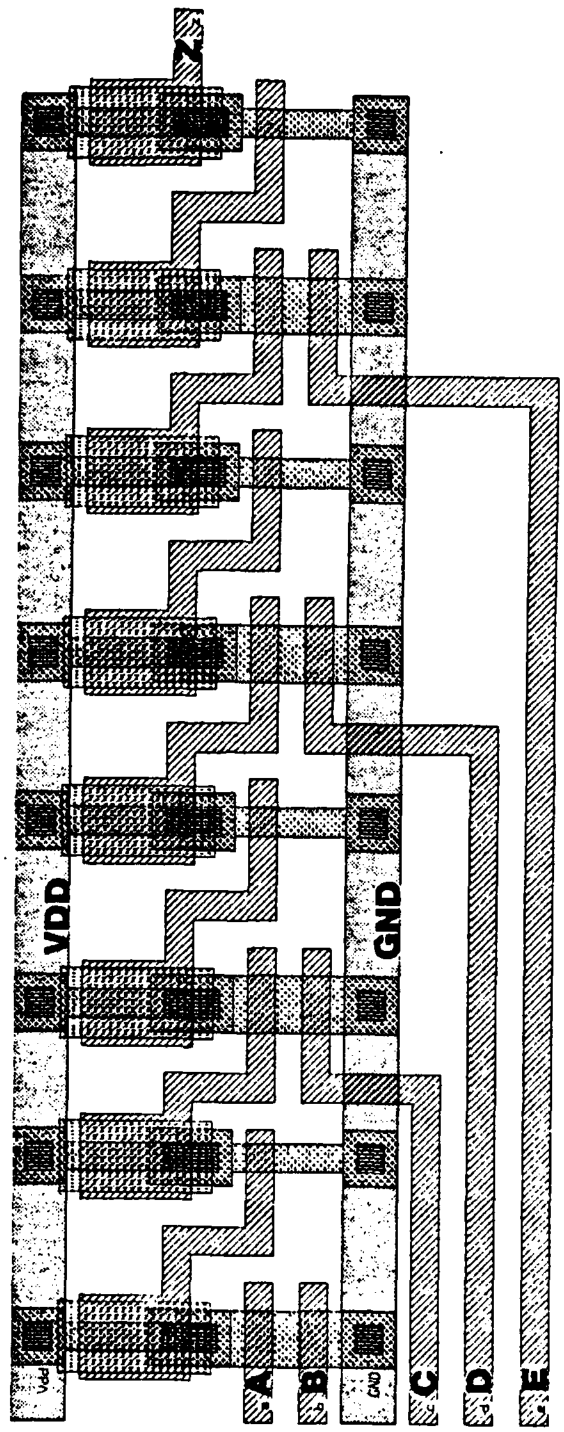


Figure 3.10 Mead-Conway Style Five Input Linear AND

patterned after this gate should be simple to incorporate into the silicon compiler. The only decisions required are how many inputs (set by the designer), spacing of the input wires (set by the design rules), and pulldown diffusion column width (must be calculated as a function of number of input wires to the gate). If a silicon compiler is desired

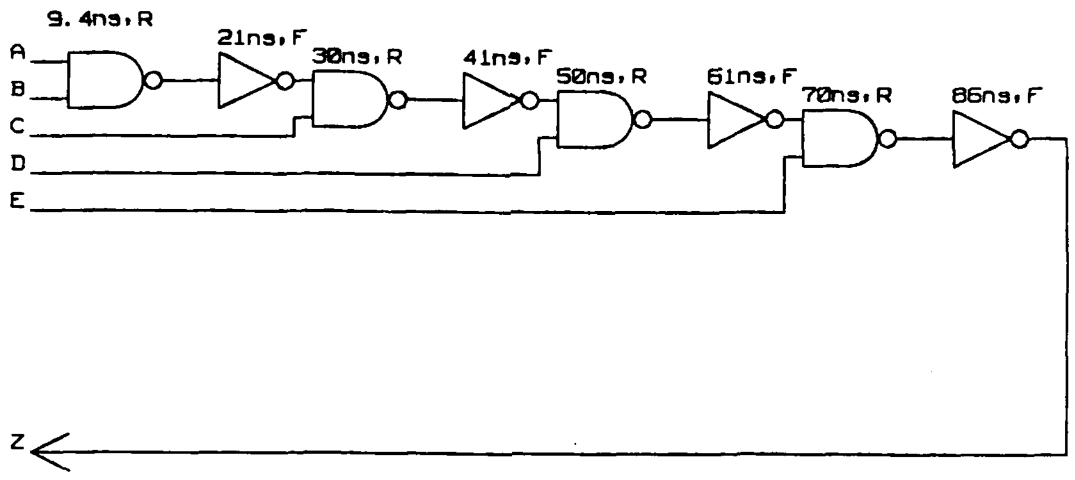


Figure 3.11 Gate Equivalent of Figure 3.10 With Delays

which produces fast, compact combinational logic circuitry, this method should be considered. Table 3.2 compares the data path AND gate (DP), the control path AND gate (CP), the hand-crafted linear cascaded AND gate (LC), and the multiple-input AND gate (MI).

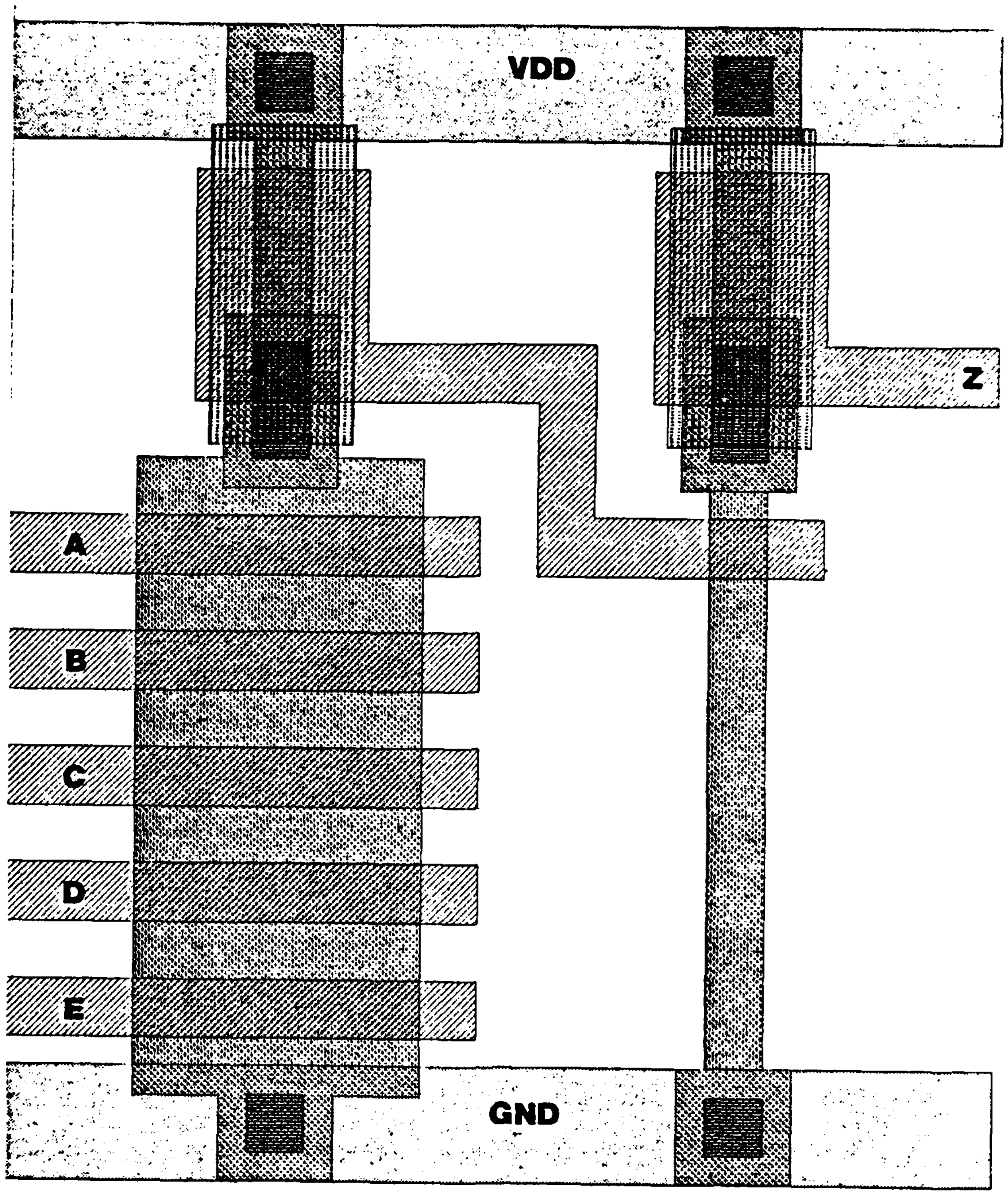


Figure 3.12 Compact Five Input AND Gate

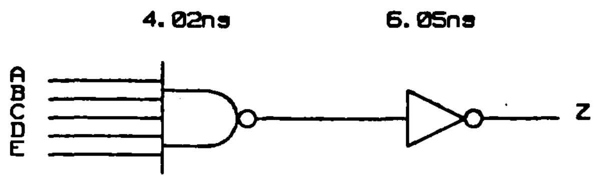


Figure 3.13 Gate Equivalent of Optimal Geometry Five Input AND Gate Showing Delays

TABLE 3.2

COMPARISON OF FIVE INPUT AND GATES

	DP	CP	LC	MI
size [mm**2]	.09	.109	.01	.004
pullups	12	8	12	1
max. pwr. [mW]	2.45	1.88	2.0	.5
prop. delay [ns]	81.15	85.98	21.97	6.05

IV. SEQUENTIAL LOGIC IN MACPITTS

Based on previous analysis, combinational logic in MacPitts is done better (i.e., more efficiently, when a choice exists) in the data path than in the control path. Does the possibility of improving MacPitts' sequential logic performance exit also? A study of this question presents interesting problems.

A. AN OVERVIEW

Chapter II discussed two different ways of increasing throughput, the PAR form and the COND form. There exists also a method of global parallelism available to the MacPitts programmer, the PROCESS form. The PROCESS form has the syntax

```
    (process  <process name>  <stack depth> ...   )
```

where the process name is an arbitrary ASCII character string (if the name is made short, then the VT-100/ADM-3A interpreter screen can display them all). The stack depth refers to the depth of subroutine calls for which this process must push return addresses onto its program counter LIFO stack. MacPitts syntax requires the designer to determine this stack depth a priori, and to explicitly state it to MacPitts (the silicon compiler). The stack depth is a required field in the PROCESS statement, and may be any integer including zero. Each process has its own stack, and all processes are executed in parallel. This parallelism provides a high throughput on a properly designed algorithm.

An extension of the digital home temperature controller of Chapter II might also control other aspects of the home environment. For instance, it would be desirable to turn tne security lights on and off by a photoelectric cell signal, to start the coffee brewing and the microwave oven cooking dinner at a timer signal, and to keep the lawn appropriately watered by turning the sprinkler on upon a moisture detector signal. The following MacPitts program outline would accomplish these tasks. All logic is done on Boolean variables, flags for storage and signals for sensor inputs.

```
(program house  <word size>

<port,signal,register,and flag assignments>

(process lite  0
     (setq lights  (not  photo_cell_input))
(process food  0
     (cond
          (six_am
               (setq mrcoffee t))
          (seven_am
               (setq mrcoffee f))
          (four45_pm
               (setq put_dinner_in t))
          (five_pm
               (setq microwave_on t))
          (five30_pm
               (setq microwave_on f)) )

(process environ 0
     (cond
          (hot
               (setq fan_on t)
               (setq window_open t)
               (setq doors_open  t))
          (cold
               (setq heater_on t)
               (setq window_open f)
               (setq doors_open f))
          (t
               (setq heater_on f)
               (setq fan_on f))
               (setq window_open t)
               (setq doors_open t)) )

(process grass  0
     (setq sprinkler_on  (not lawn_moist)) )

(process clock  1
     (par(call mod60)(setq time counter_out)) )

mod60
     <a   modulo  sixty  up  counter   algorithm>(return))
```

All of these processes are done in parallel. All of the processes have a stack depth of zero except for the clock process, which has a stack depth of one. This is necessary due to the clock process calling a subroutine, the modulo sixty up counter. The call of the counter and the following SETQ are paralleled with the PAR construct. This PAR paralleling appears to work well for cases where the output depends on the called routine, like the example above. If the dependency is reversed (for instance, paralleling SETQs of inputs to a slow multiplier subroutine with the CALL to that multiplier) some unpredictable results can arise. A good practice is to emulate all time-dependent algorithms alone in the interpreter prior to their incorporation into the MacPitts algorithm. In so doing, syntax errors may be found and fixed and the algorithm may be optimized for number of cycles required to execute.

For fast architectures, some additional speed can be gained by paralleling the subroutine outputs with the RETURN from the subroutine. For instance, the mod60 counter-timer in the previous example is called as a subroutine.

```
mod60
   .
   .
   .
(par(setq counter_out count)(return))
```

There exists no time-dependency between the final result (counter_out) and the RETURN to the main program, so no data latency results from this paralleling.

To re-emphasize, all of the PROCESSes under the PROGRAM statement execute in parallel. So while the <house> chip is monitoring temperature and time, it is simultaneously monitoring lawn moisture, setting the house clock, and checking the outside light level. PROCESSes execute independently, in parallel. Each PROCESS has its own independent stack, and processes do not communicate internally with each other. From the hardware standpoint, each process is an independent MacPitts entity sharing data storage elements and signal wires.

In this somewhat artificial example, there is no strict requirement for speed. If the lawn is watered 50 microseconds late, the grass will still grow. But the principle of global process parallelism applies to more complicated digital systems where intricate timing interrelationships exist. It is also evident that MacPitts is a very versatile silicon compiler. A chip constructed from a similar multi-process algorithm could be used to control many off-chip processes simultaneously. The intrinsic nature of the PROCESS form lends itself well to applications such as industrial digital control. In situations where the PROCESS statement is used to force parallelism but the parallelism is not needed (for instance, the <house> algorithm), MacPitts creates a large layout. Silicon area is traded off for speed.

This algorithmic outline illustrates using PROCESSes in a combinational logic machine. PROCESSes are required around any invocation of a subroutine, but aside from this consideration, the <house> chip could be specified just as well without PROCESSes.

PROCESSes are required, however, to describe a sequential logic machine in MacPitts. The FSM architecture is explicitly specified by the PROCESS form. The PROCESS statement implicitly specifies creation sequencers (a data path hardware organelle, which steps the FSM through its states) and their instantiation in the data path.

B. GRAY CODE TO BINARY DECODER

The following section illustrates the MacPitts design of a simple sequential logic system. The Gray code [Ref.5: p.97] finds many diverse uses in electrical engineering and computer science. Whenever a single bit change in successive data words is desired, (disk sector addressing, radar antenna positioning) the Gray code should be considered. In finite automata theory, the Gray code decoder can be regarded as a sequence detector. The desired sequential machine complements the input on having received an odd number of earlier 1's, and does not complement the input on an even number of 1's. An example sequence is

```
  input:  1 1 1 1 0 0 0 0 1 0 1 0 1 1 0 0 1 ...
  output: - 0 1 0 0 0 0 0 1 1 0 0 1 0 0 0 1 ...
```

The Gray code decoder can be implemented in MacPitts as a Mealy FSM to detect this sequence, and set the appropriate outputs. The automata for the Gray code decoder is shown in Figure 4.1. The node label MSBS indicates most significant bits, COMPL means complement the present bit, and NEXTBIT means consider the next bit.

1. Algorithm Design

The next consideration is algorithm design. Previous experience inclines the designer toward a data path architecture (faster, smaller, less power consumption). Furthermore, a data path chip would probably have a greater throughput, since the operations could be done on words, and

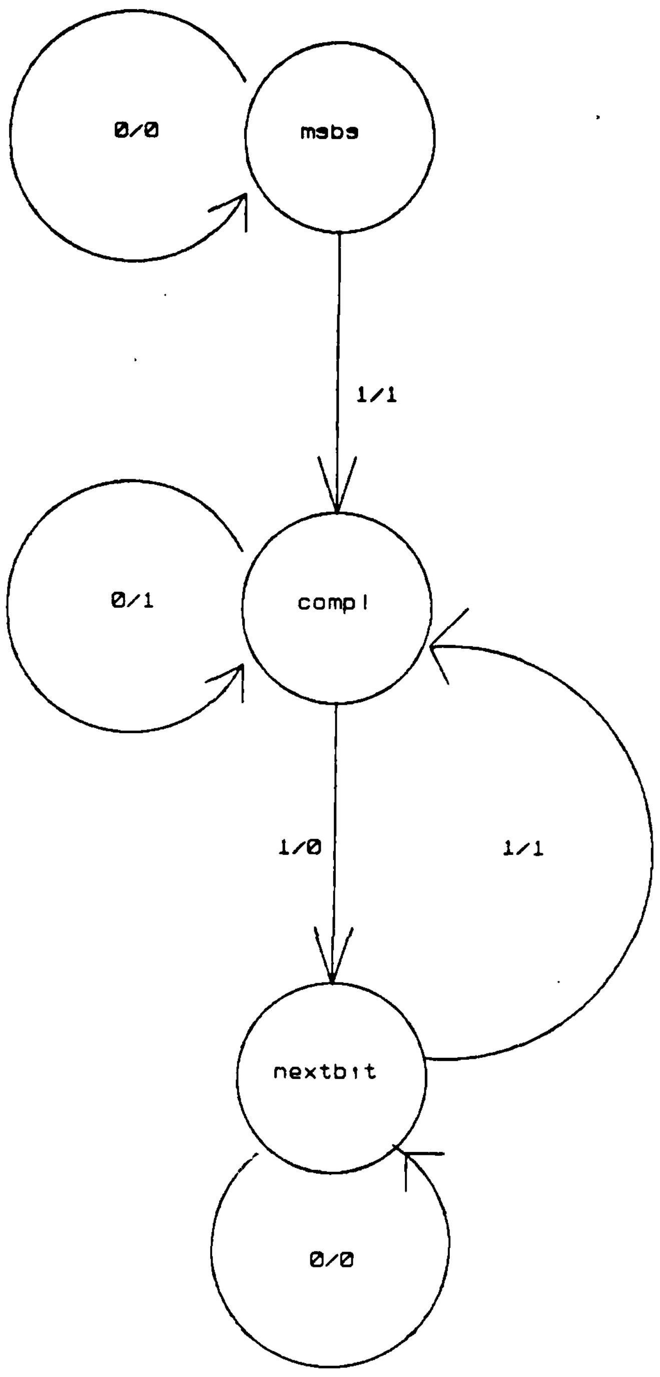


Figure 4.1 Gray Code Decoder State Transition Diagram

not individual bits (e. g., a parallel Gray code decoder, which decodes on a word basis rather than a bit-by-bit basis).

The problem with this approach is that MacPitts permits no explicit, succinct method of setting the individual bits in a word. The bits can be tested with the BIT expression, but not set. So a control path (implying Boolean type data and Weinberger array combinational logic) architecture is probably a better choice.

A control path FSM can be designed with MacPitts (even though no explicit data path is used). The reason is the way in which MacPitts implements FSM state transitioniong with the sequencer organelles. The sequencer can be thought of as a bank of n sequencer organelles, where n is the data path width specified in the PROGRAM statement. The sequencer organelles are physically adjoined to the data path organelles in the MacPitts chip. The sequencer stores FSM state, much in the same way as flip-flops store state in a discrete-chip FSM design. And just as two raised to the power (number of flip-flops) limits the states in a discrete digital system, so two raised to (number of sequencers) limits the states possible in a MacPitts sequential machine. The number of sequencers is always equal to n, the data path width. This has ramifications for MacPitts designers considering a system of many states with a narrow data path. The possible number of states is limited to 2**n.

One solution to the Gray code problem is to use a data path architecture, to declare the data path width as two, and to specify an extra (unused) bit in the input and output port declaration statements. The most significant bit of the input port is obviously extraneous, but the data path width of two is necessary to address the three states required (Figure 4.1). When the Gray code chip is used, these extra pins must be tied to ground. If a data path width of one is specified (and PORTS are used for inputs) instead, MacPitts gives the following diagnostic

```
Error-Word length too small to store the state for
this process
```

If the data path width is left as two, but the input and output ports are left only one bit wide (another attempt to circumvent this problem), MacPitts responds with

```
Error-Invalid port definition
```

which means that the data path width was declared as two, but the port is only one bit wide (MacPitts has helpful diagnostics). The MacPitts source code file, extract.lisp (under the def get-sequencer-from-process macro) shows why this constraint exists. The sequencer width is explicity set to the data path width.

Figure 4.2 shows the MacPitts driver code to do the Gray code to binary conversion serially. The MacPitts algorithm shown in Figure 4.2 has the lines numbered for reference, but the numbers are not part of the allowed MacPitts syntax. Line 1 is the title, using a semicolon as the reserved word comment designator. Line 2 is the PROGRAM statement, the program name is gc (Gray code) and the data

```
1   ;Grey Code to binary conversion algorithm
    ;This code illustrates the Data Path (i. e.,
    ;Integer) solution to the problem. It is but one
    ;Variant of many possible solutions.
    ;Define the data path width as 2 (state transitioning)
2   (program gc 2
3   (def 1 ground)
4   (def 2 phia)
5   (def 3 phib)
6   (def 4 phic)
    ;All FSMs must have a RESET input (for initialization)
7   (def reset signal input 5)
    ;Use INTEGER (port) input & output, 2 bits wide
8   (def inp port input (6 7))
9   (def bin port output (8 9))
1Ø  (def 1Ø  power)

    ;Specify FSM architecture
11  (process grycod  Ø

12  msbs   ;  (Most Significant Bits)
13      (cond((=Ø inp)(setq bin Ø)(go msbs))
14            ((= 1 inp)(setq bin 1)(go compl)))

15  compl  ;  (COMPLement bits)
16      (cond((=Ø inp)(setq bin 1)(go compl))
17            ((= 1 inp)(setq bin Ø)(go nextbit)))

18  nextbit;  (NEXTBIT in string)
19      (cond((=Ø inp)(setq bin Ø)(go nextbit))
2Ø            ((= 1 inp)(setq bin 1)(go compl)))  ) )
```

Figure 4.2 Gc.mac

path width is two. Lines 3, 4, 5, 6, and 10 are standard, and required by MacPitts conventions. Line 7 is required for all FSMs, and when it is raised high (positive logic arbitrarily chosen here), the FSM/PROCESS is reset to its initial state. Line 8 defines the input port, inp, and declares it integer two bits wide. Line 9 does the same for the output port, bin (binary value). Line 11 specifies FSM architecture with the PROCESS statement, for which the stack depth is zero (no calls to subroutines). Line 12 is a node label, msbs (most significant bits), and represents the top node in Figure 4.1. Line 13 is the first check in this state, and says that if the input equals zero, then set the output to zero and go to node msbs. If the input does not equal zero, then go to the next line of code. Line 14 checks whether the input equals one. If the input is equal to one, the output is set to one, and the program transitions to the complement (compl) state. Line 15 implements the second node in Figure 4.1, complementing the input. Line 16 checks the input, and if it equals zero it complements and keeps complementing as long as the input equals zero, and if not, it proceeds to the next line. Line 17 checks for the sequence of an even number of ones, and if true, sequences to the next node after complementing the input. Line 18 is the label corresponding to the last node in Figure 4.1, nextbit. Line 19 checks the input bits, sets the output to the input value, and returns to this node as long as the input is zero. Line 20 also sets the output to the input value, but jumps back to the bit complement node when the input is one. The conditional in line 17 is unnecessary, but is included for clarity (If the non-storage port, bin, is not explicitly set to one, it will become zero at the next state transition. Line 17 can be eliminated, and the algorithm will work correctly anyway).

The next step is to test and debug the algorithm in the interpreter prior to full compilation. The Gray code algorithm was debugged in the interpreter, and compiled with the <herald> option. Appendix B shows the script recording of the compilation process, and indicates a data path of seven different organelles (to be discussed in the next section) and a moderate-sized (31 columns) Weinberger array.

Figure 4.3 shows the chip resulting from the compilation of gc.mac. The functional constituents of this layout will be treated qualitatively in the next section.

2. Functional Constituents Of The Chip

The layout scheme of MacPitts places general functional blocks in specific relative locations on the chip. Figure 4.4 indicates where these relative locations lie on the cifplot. The block sizes shown in Figure 4.4 are arbitrary, since the actual sizes depend on a combination of algorithm and MacPitts (the source code). In comparing Figure 4.4 to Figure 4.3, it is seen that this chip has no flags, which is expected since none are defined in the source algorithm. The rest of the blocks shown in Figure 4.4 are instantiated in cg.cif (Figure 4.3).

The data path arithmetic block is shown in Figure 4.5. The function of this unit is to operate on the inputs

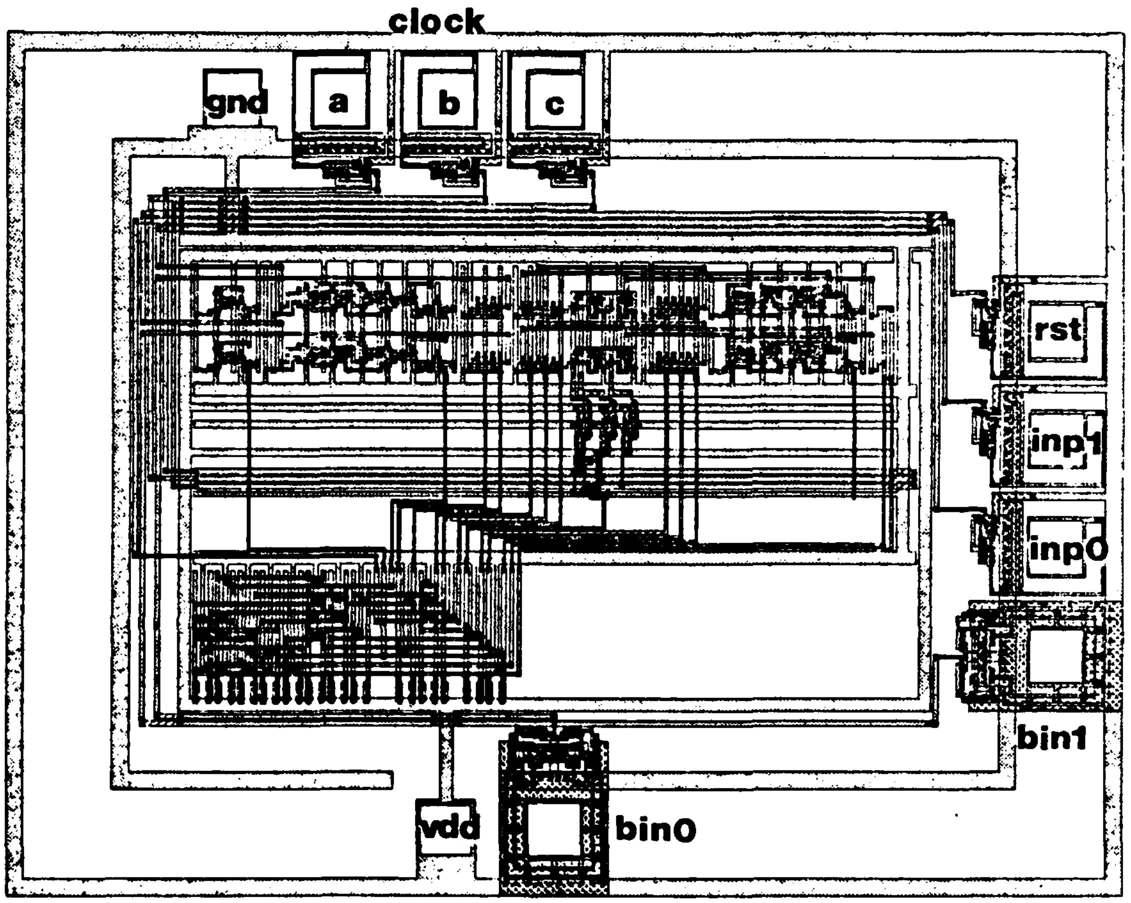


Figure 4.3 Gc.cif

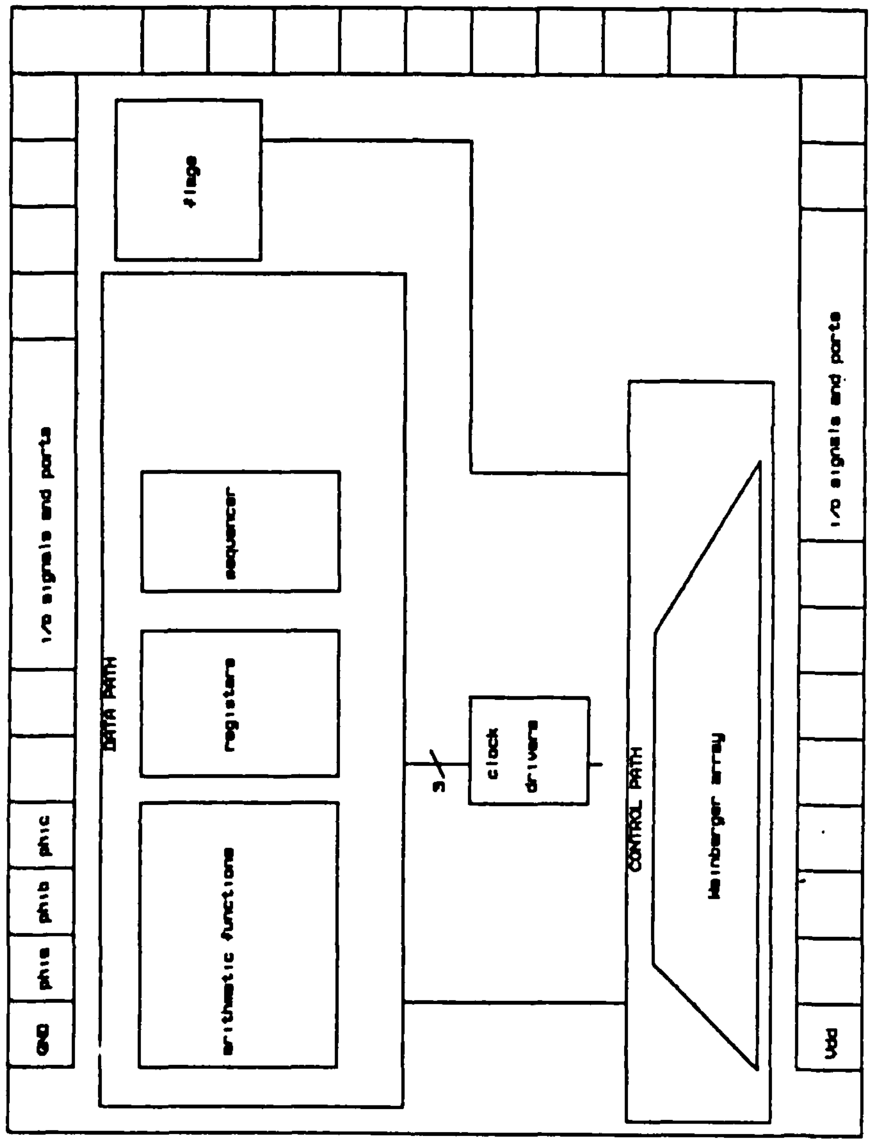


Figure 4.4 MacPitts Layout Scheme

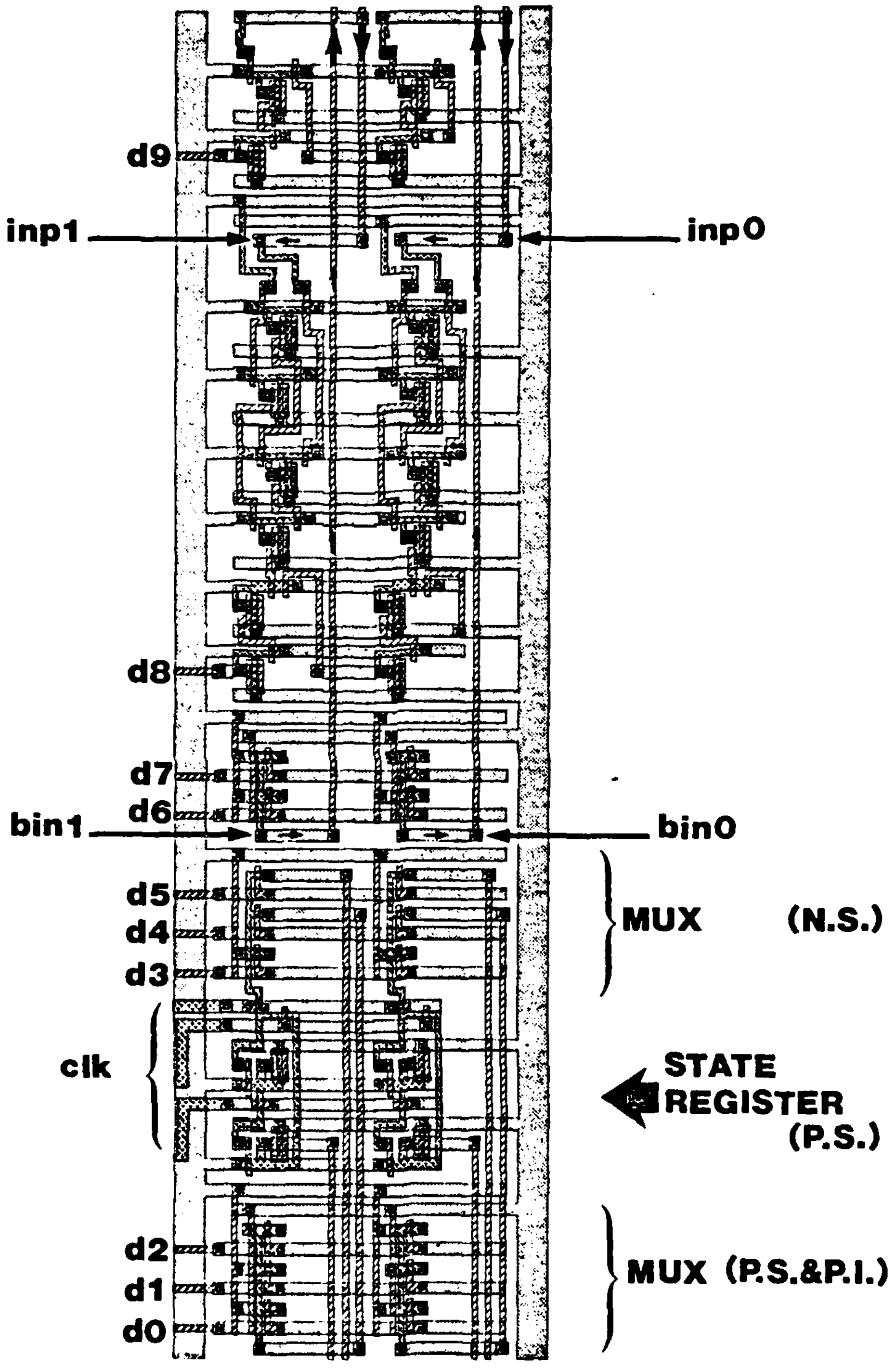


Figure 4.5 Data Path Arithmetic Block From GC.cif

so that the desired outputs result. The inputs enter the arithmetic block and the outputs exit as shown in Figure 4.5. Between input and output, the data is subject to switching and various logic operations. The data path and the control path must also communicate with each other over the interconnecting traces. The leftmost top poly line, D9, is an input to the Weinberger array, where it turns on five NOR gates. Similarly, the other nine lines also connect to the control path. Lines D8, D7, D5, D4, D3 (reset), D2, D1, and D0 are outputs from the control path and inputs to the data path. Line D6 is the other output from the data path to the control path. The inputs to the data path can be understood as relay controls, or switches. The outputs from the data path to the Weinberger array are Boolean values to cause decisions about what to do next.

From Figure 4.5, the arithmetic path of this chip is seen to be two bits wide (the two horizontal parallel organelle chains). In Chapter II it was shown that syntax implicitly controls instantiation. Line 13 in the Gray code algorithm specifies two data path operations

```
(cond((=0 inp)(setq bin 0)
```

where the (=0 inp) is a logical comparison integer test, and (setq bin 0) is an integer form by definition of bin in the def statement and the source for bin being an integer, zero. The leftmost set of cascaded OR gates makes the (=0 inp) test, and signals the control path on line D9. Figure 4.6 shows the logic diagram for this stipple plot, and the results for a zero input.

Proceeding right on the arithmetic block stipple plot, the next block is a set of paralleled NOR gates. The inputs are the inp bits, inp0 and inp1, and Vdd and GND. The output is a signal to the control path from D8 which determines the chip output, bin (BINary equivalent of the Gray code bit stream). This circuit does not directly make the output assignment, (setq bin 0), but rather does it through combinational logic in the Weinberger array. Figure 4.7 is the logic diagram of the setq operation circuitry. The circuit is annotated to show a zero bit input on inp, in which case a TRUE is sent to the control path on line D8.

Proceeding right in the data path, the next two blocks in Figure 4.5 show pass transistor units. The leftmost pass transistor unit has inputs from bin0, bin1, and control on D7. The output is a signal to control on D6. This section of the data path is where the output bin is set, although the logic for setting bin is determined in the preceeding two data path units and the control path. To the right of this unit is another pass transistor block which takes inputs from the previous pass transistor unit, from the clock drivers, from control on lines D5, D4, and D3, and from the sequencer. The function of this unit is state transition. The sequencer inputs represent the current

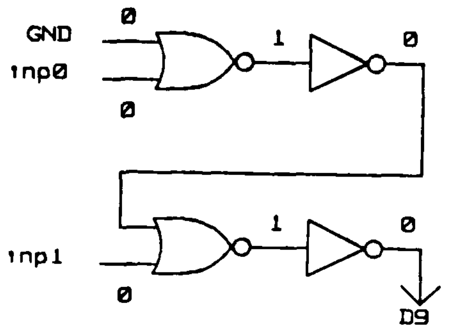


Figure 4.6 Test Logic for (=0 inp)

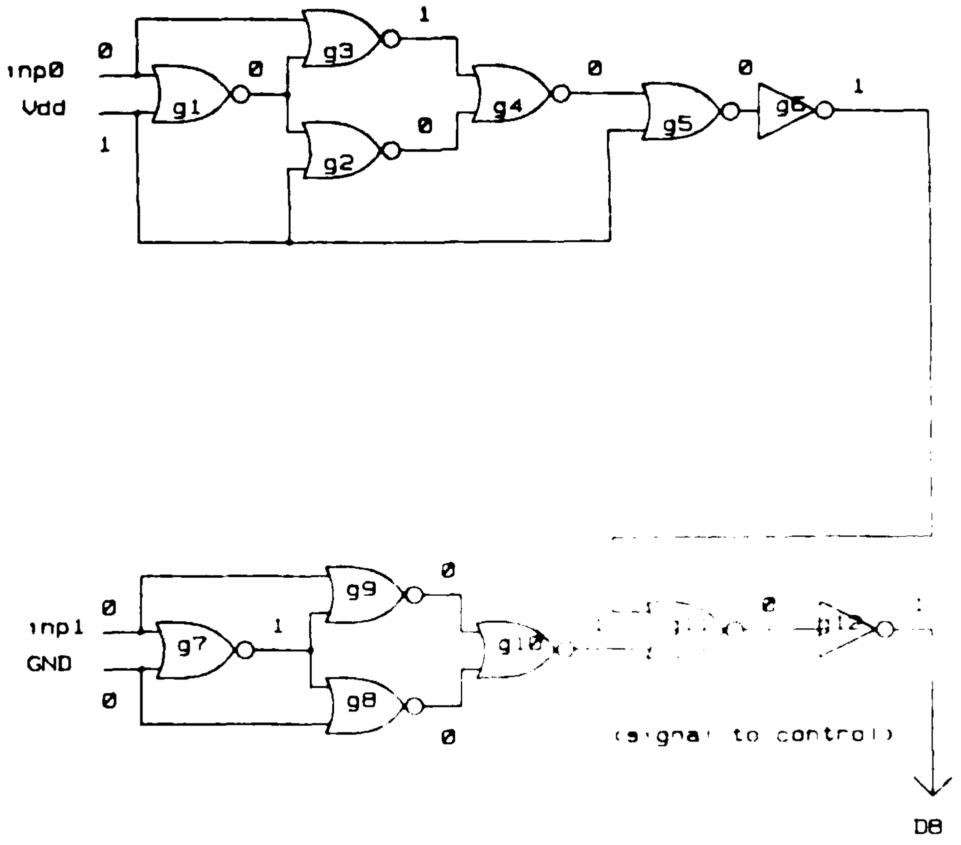


Figure 4.7 SETQ Operation (Signal to Control)

state, and this unit drives the state registers which signal next state to the sequencer tail, at far right. The input D3 is the reset signal, which implements the MacPitts function of returning the FSM to its initial state when raised high.

Figure 4.8 shows the state registers, a set of parallel 2-T memory cells, in which the current state is held. The inputs to the state registers are the outputs of the previous pass transistor block, signalling next state transition, and the three clock lines from the clock driver. The outputs are the two state bits (S0 and S1) to the control path (on lines marked C1 and C0, Figure 4.10). The Mealy FSM methodology is evident in MacPitts from both the algorithmic and hardware viewpoints. The output is a function of both input (inp0, inp1) and present state (S0, S1).

Below the state registers in Figure 4.3 are the clock drivers. Figure 4.9 is a blowup of the driver organelles, used for buffering the clock signals and generating the five overlapping clock signals. The drivers are turned on by a signal from the Weinberger array, C5. Carlson describes the clocking scheme and the reasons behind its choice (Ref. 2:p. 26).

The rightmost block in the data path is the sequencer. Figure 4.10 is the cifplot of the sequencer combinational logic, and Figure 4.11 is its gate equivalent. The sequencer has as its inputs the current state (S0, S1)

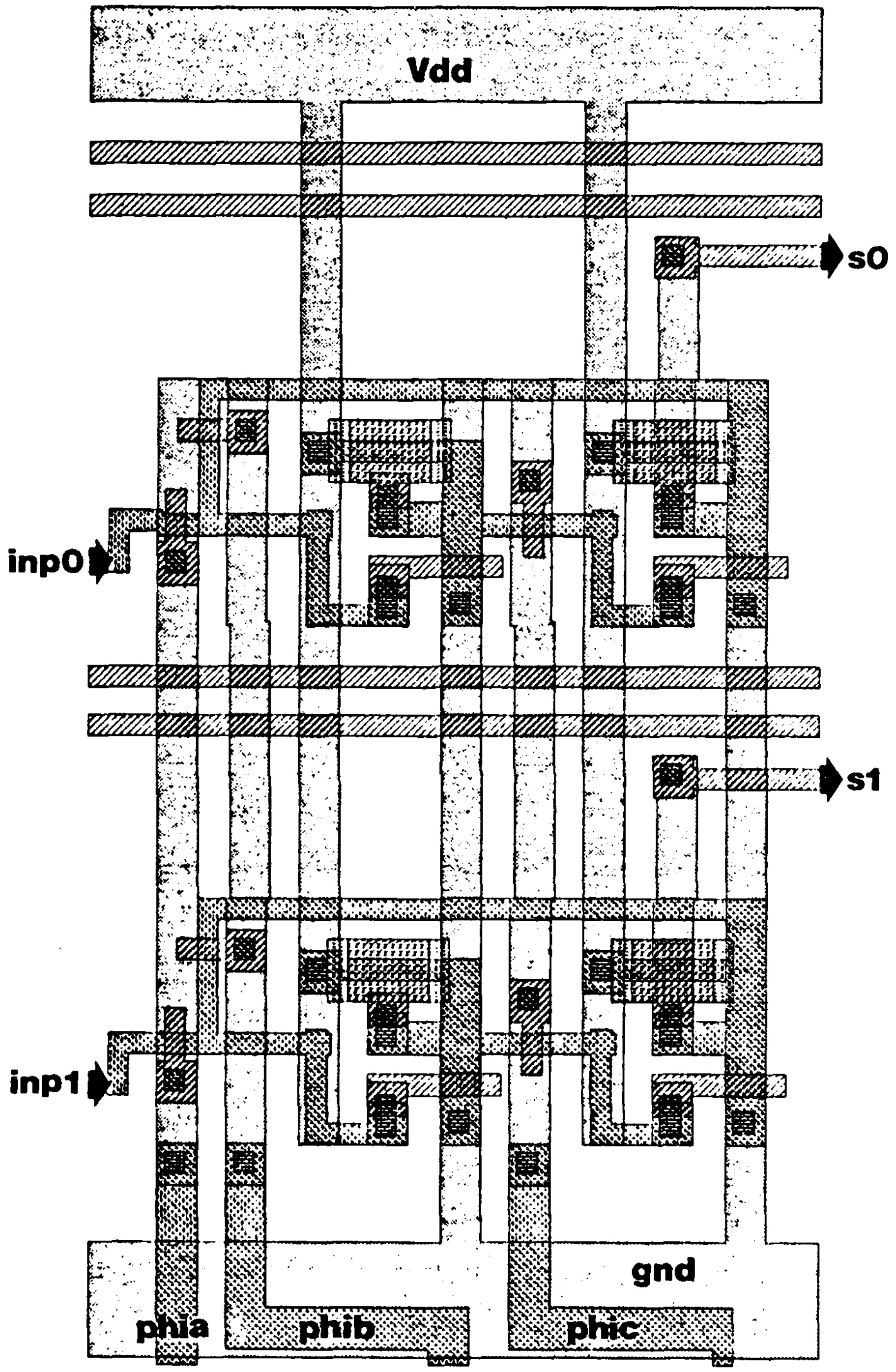


Figure 4.8 MacPitts State Registers

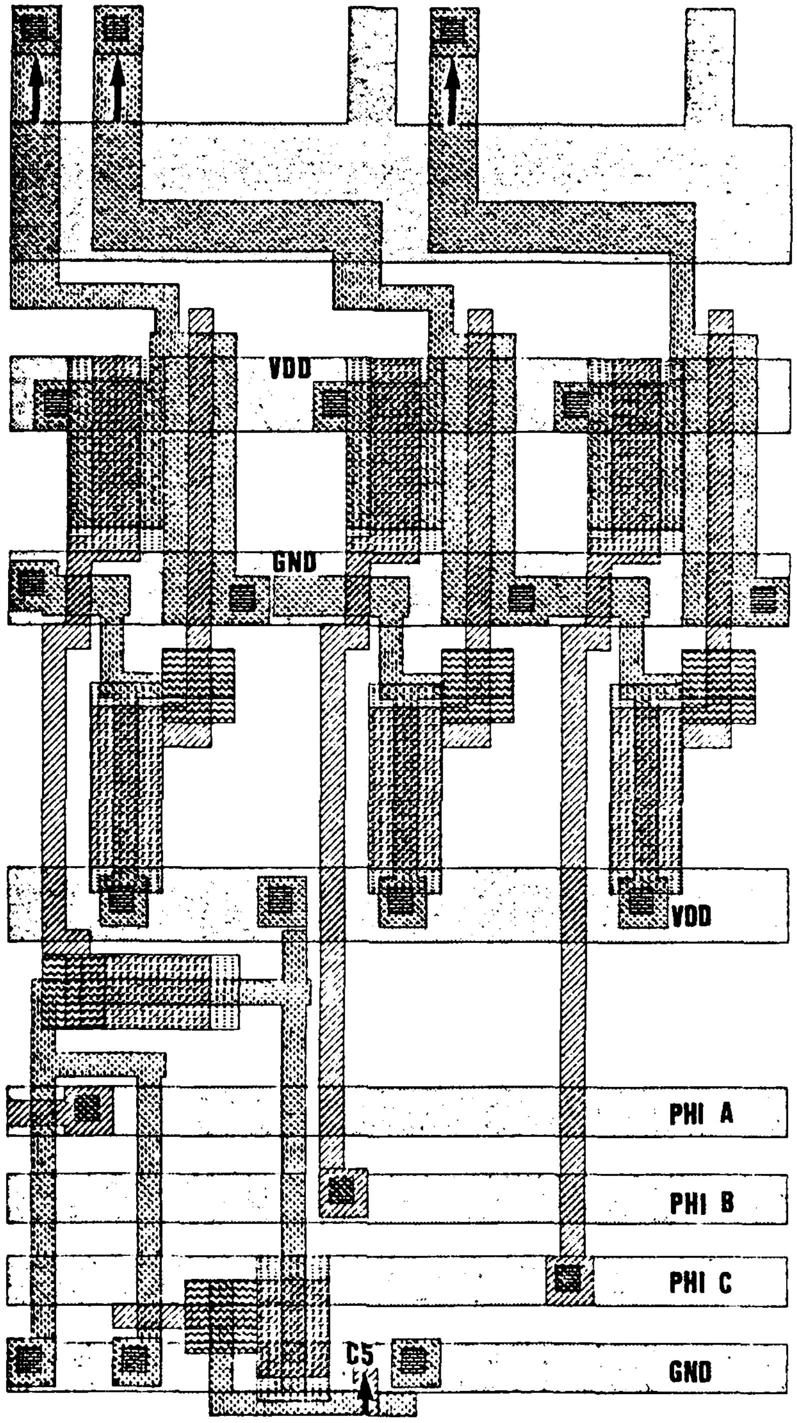


Figure 4.9 Clock Drivers and Five Segment Generator

and produces as its outputs the next state (S0+1, S1+1). The gate diagram of the sequencer answers the question asked in the initial design of the Gray code decoder, why three states are not allowed in a control path (i.e., a data path width of zero) architecture. The answer lies in the implied data path structure, as explained previously and as graphically shown in Figure 4.10 and Figure 4.11. The data path width as specified in the PROGRAM statement sets the number of sequencers to be instantiated, and the number of sequencers limits the number of states possible. If fewer FSM states are required than the sequencer depth can transition to, the sequencers are nevertheless instantiated, but their outputs are not connected to the control path (C0 and C1 in this example). For example, this would occur for a wide data path which had few states. If a data path FSM chip were designed with a word length of five , and only four FSM states were needed, MacPitts would instantiate all five of the sequencer organelles. Only the top two would be connected to the Weinberger array. Figure 4.12 is a block diagram of the MacPitts sequencer organelles, and shows how the Mealy FSM is implemented. The multiplexers on each side of the state registers determine that the next state is a function of both present state and present input. The Weinberger array controls the gating in the multiplexers to allow the appropriate signals to pass to the state registers.

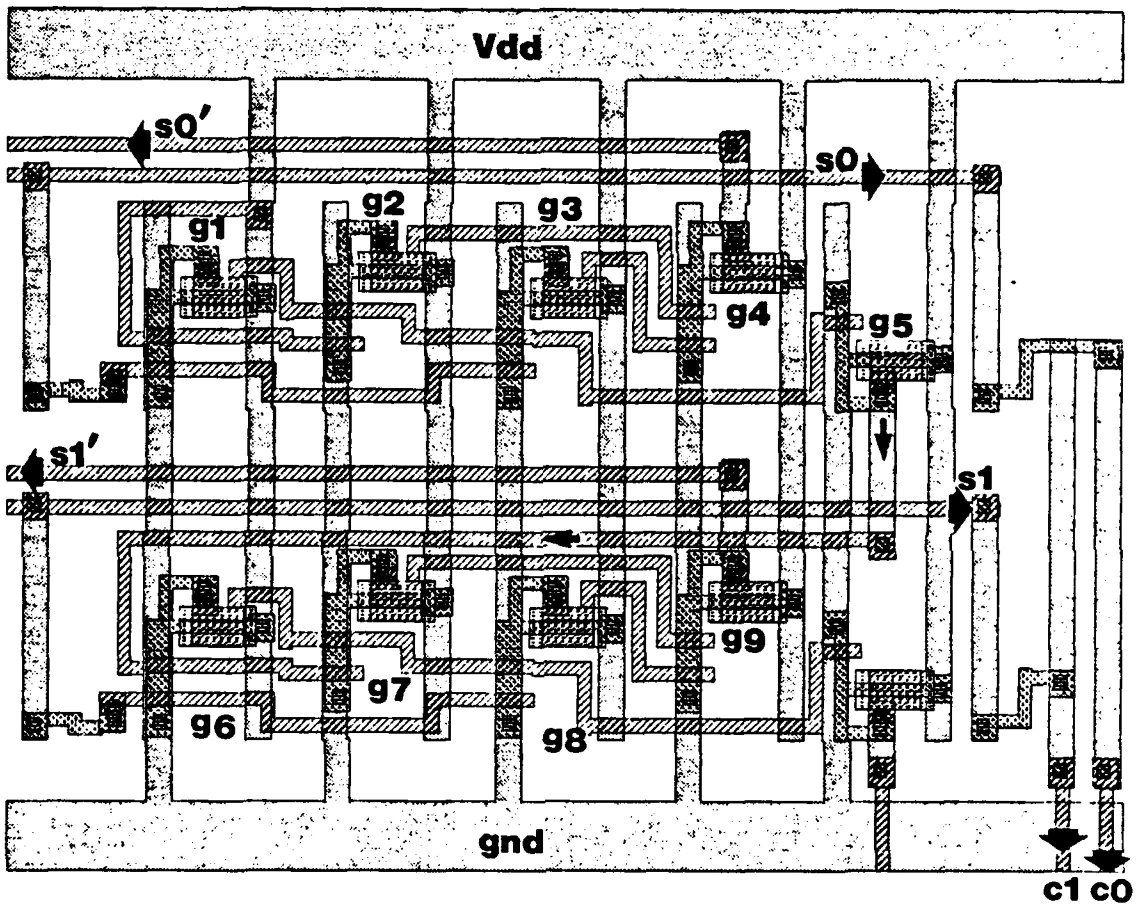


Figure 4.10 Sequencer Tail

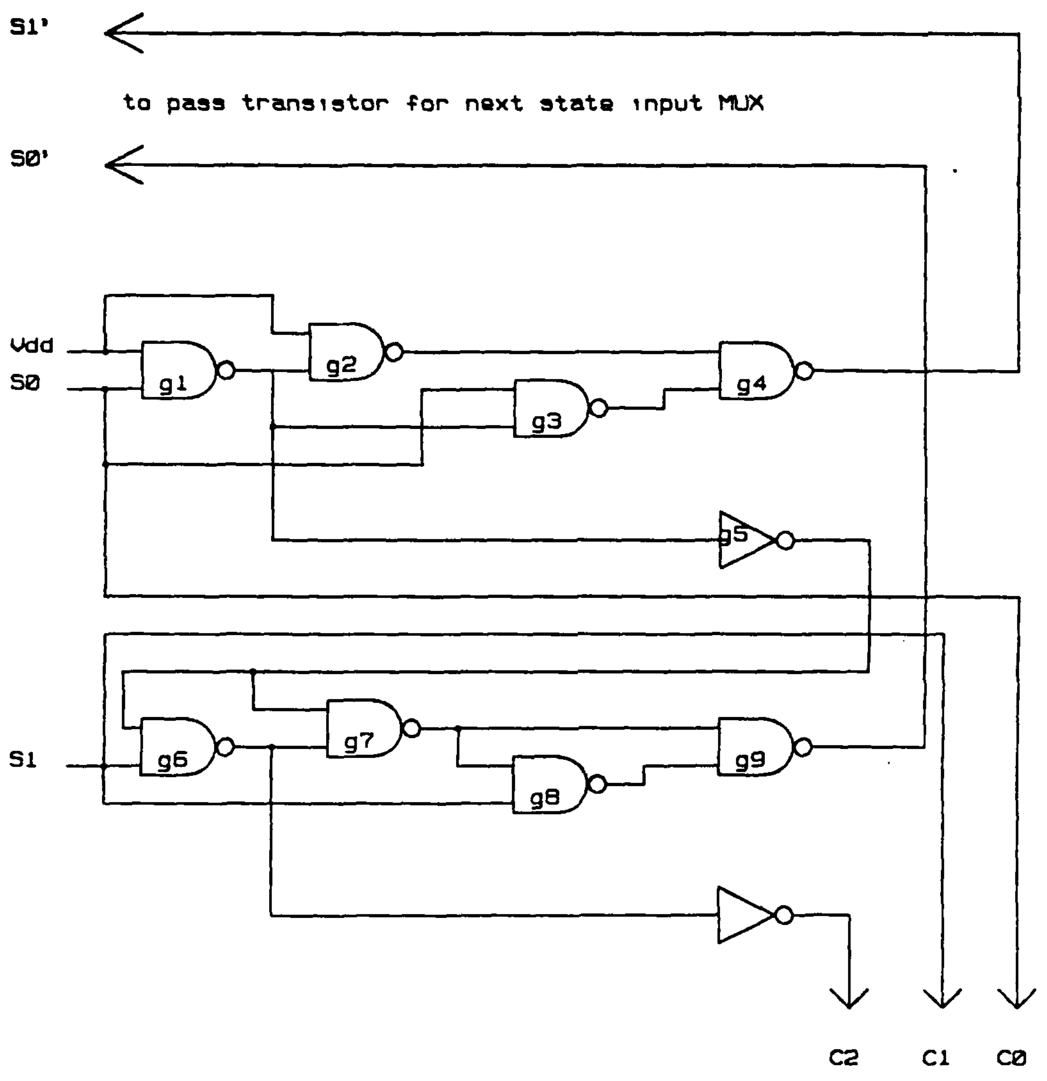


Figure 4.11 Sequencer Tail Logic Diagram

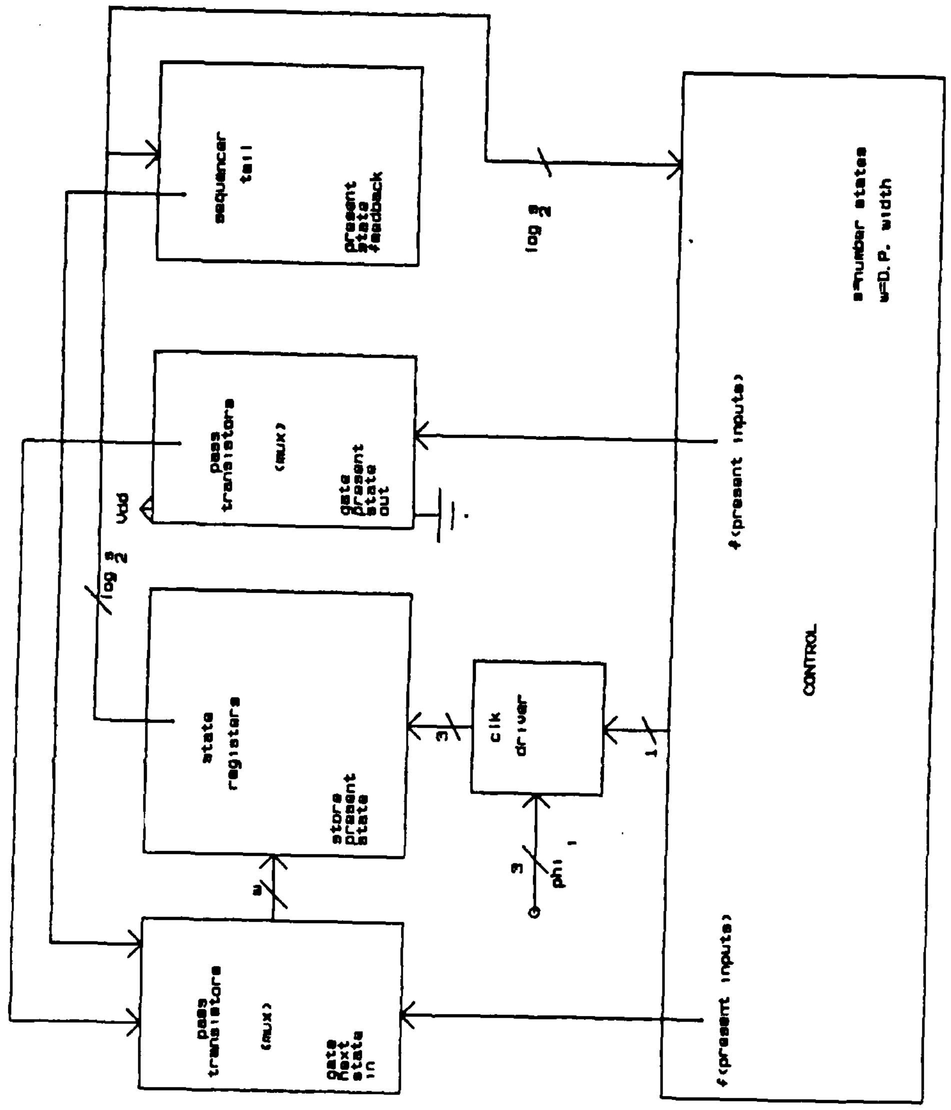


Figure 4.12 Mealy FSM Implementation in MacPitts

The Weinberger array is the control path in a MacPitts chip, for reasons explained in Chapter II. The Weinberger array is shown in Figure 4.13, and its labelled gate equivalent is Figure 4.14. In the cifplot, all input and output columns have been labelled (A-Z) for comparison. The output lines have also been labelled (Cn) for reference to the other functional blocks of the chip. There are major differences between this multiple function Weinberger array and the single function Weinberger arrays considered previously.

This Weinberger array for single output functions always has a four level structure, inverter-NOR-NOR-inverter. This is not the case for multiple output Weinberger arrays. This circuit has 11 inverters and 15 NOR gates. The maximum fan in on any NOR gate is six. In the previous Weinberger arrays, the maximum delay was approximately four gate delays. In this Weinberger array, the longest path is shown in Figure 4.14 as J-U-T-L-F-G-D, or Q-W-T-L-F-G-D, Each path induces approximately seven gate delays. The MacPitts script session (included in Appendix B) lists the control depth (NOR gate nesting) incorrectly as four. Furthermore, the polysilicon runs cover proportionally more area in this Weinberger array than in the previous single function ones. From Chapter III, the polysilicon to substrate capacitance is a strong factor in limiting chip speed. The multiple function Weinberger arrays are expected

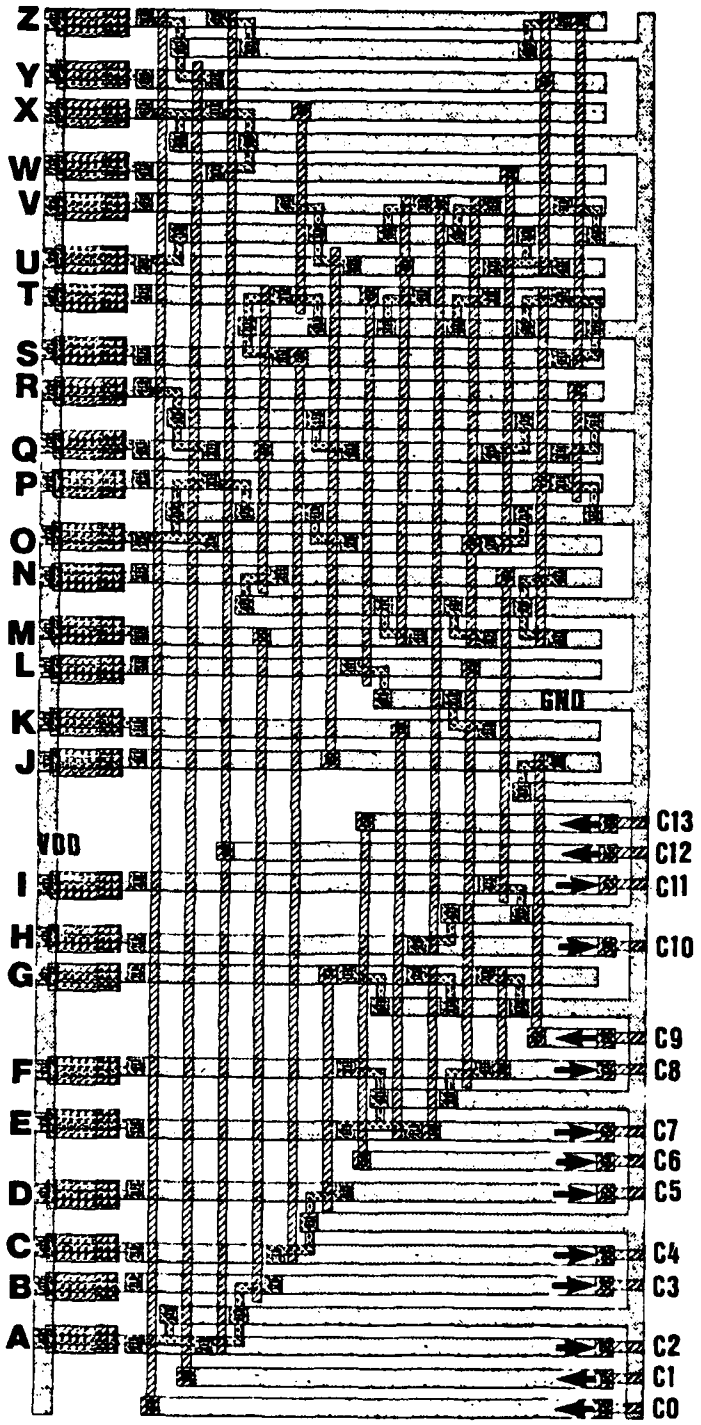


Figure 4.13 Weinberger Array from Gc.cif

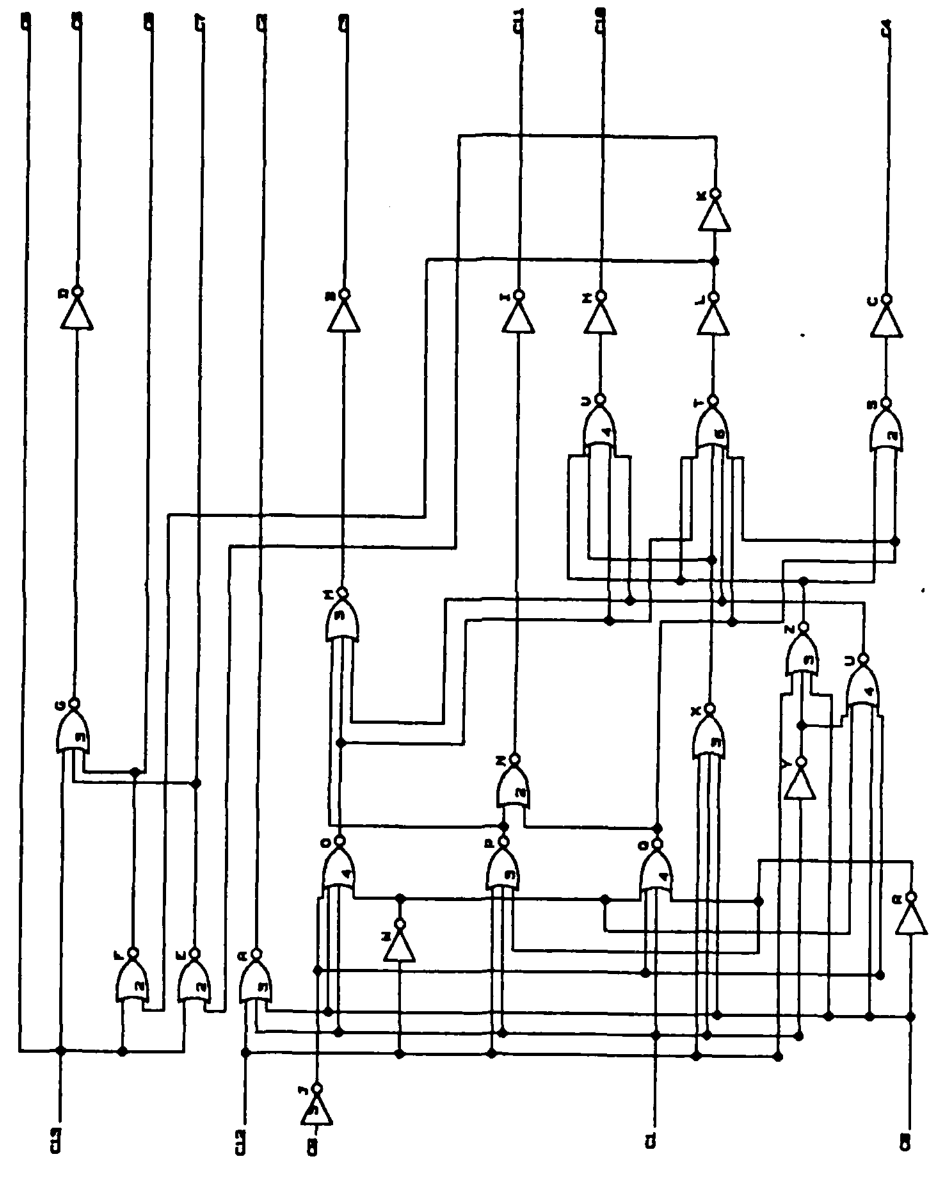


Figure 4.14 Gate Equivalent of Gray Code Weinberger Array

to be slow. This Weinberger array has nine outputs (C11, C10, C8, C7, C6, C5, C4, C3, and C2) and five inputs (C13, C12, C9, C1, and C0). C13 is a check on the input signal values, and comes from D9 (D indicates a signal to or from the data path, C indicates a signal to or from the control path). C11 is an output to D8 in the data path, the function of which is not clear (data path output connecting control path output). C10 is an output to D7, and the signal controls pass transistor gating in the left pass transistor unit, which determines the value of the output (bin0, bin1). C9 is an input to the Weinberger array, and comes from D6. This input is not set within the data path, and it is likely that it results from MacPitts' expectations of a more complicated structure. The sequencer organelles exhibit this vestigal structure property also, as previously mentioned. C8, C7, and C6 are outputs which control the second pass transistor block (state register multiplexer) in the data path. They connect to D5, D4, and D3, respectively, and control the sequencer's next-state transitioning. C6 is connected to pin five by a polysilicon run and C13, so C6 (D3) is the reset signal. C5 is an output which turns on the clock drivers. C4, C3, and C2 are outputs connecting the data path at D2, D1, and D0, where they control pass transistor gating for the sequencers and state register. C1 and C0 are inputs from the state register which represent the current state. Figure 4.15 shows the data path-control path interconnections. The interconnections are summarized in the diagram below.

	(inp)	(bin)	PT	ves	PT: state				PT:seq and reg				
rst	D9	D8	D7	D6	D5	D4	D3	clk	D2	D1	D0	S1	S0
C13	C12	C11	C10	C9	C8	C7	C6 rst	C5	C4	C3	C2	C1	C0

In this diagram, rst means reset, PT is a pass transistor unit, ves is a vestigal (non-functional) unit, seq is sequencer, and reg is the state register.

3. <u>Alternate Designs</u>

The gc.mac algorithm used explicit value assignment in the output setq forms.

```
  (setq bin <value>)
```

In this case, it is possible to explicitly set the output to a value (one or zero). This is not possible, however, for all algorithms, and is not even desirable in the general case. Usually the output is a function of the input(s), and not a specific value which is known beforehand. With this in mind, an alternate algorithm was written to implement the Gray code to binary conversion. Figure 4.16 shows the algorithm, gc2.mac. This code follows the state diagram given for gc.mac (Figure 4.2), and the states all have the same names. The algorithms are equivalent functionally and semantically (they both do and

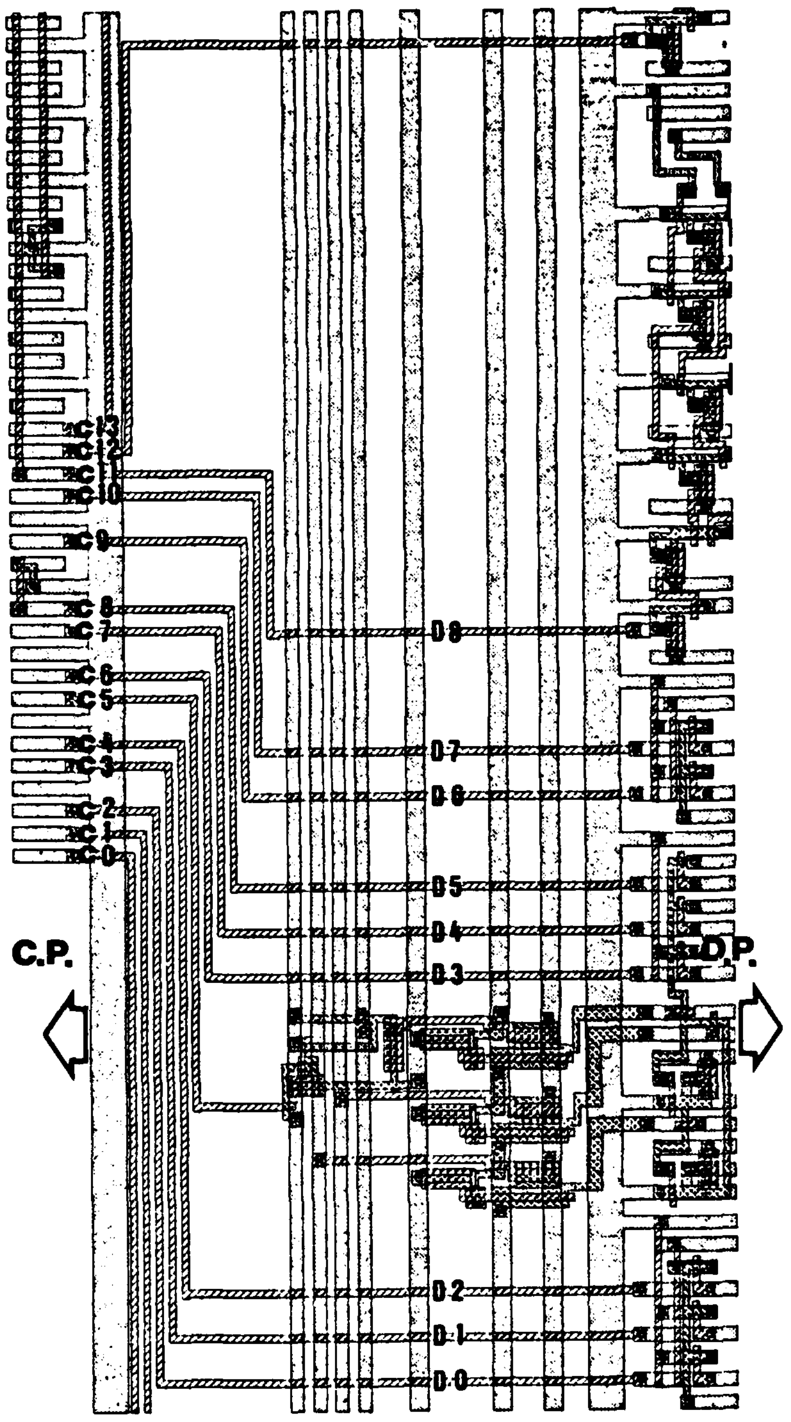


Figure 4.15 Data Path/Control Path Interconnection

say the same thing). The only difference is in the binary output (bin) setq forms. In the previous algorithm, gc.mac, the output bin is explicitly set to either one or zero. In this algorithm, gc2.mac, the output is set to a data path function of the input. The code represented by gc2.mac represents the more general case.

The chip created by gc2.mac is expected to be larger than the one created by gc.mac, since additional data path decisions are required in the setq forms. The script file of the gc2 MacPitts session, (Appendix B) verifies this, and Figure 4.16 shows the resulting layout. In comparing the two script files, it is seen that gc2 requires more data path units, data path transistors, and control path transistors. This is reflected in the comparative complexities of the data paths in Figure 4.3 and Figure 4.16. The chip produced by gc2 would also consume slightly more power and be slightly larger than the chip produced by gc. The conclusion is that by explicitly specifying the setq destination values, the designer can save area and power consumption. A reasonable expectation would also be a faster chip.

Explicit assignment of outputs is therefore desirable, though not always feasible. In many control path architectures, where the output is treated as individual bits, explicit assignment is possible (though not always the optimal solution, see Chapter VI on Hamming error correctors, where there are many outputs possible). In data path or hybrid architectures where there are only a few numerical outputs possible, explicit assignment of output values should also be considered (see the blackjack algorithm, following). A general rule is to choose the method that results in the shorter algorithm, whether (1) explicit assignment of outputs, or (2) assignment of outputs as a function of either inputs or intermediate values. The significance of this is that the designer can influence the design by the MacPitts program written, even though the silicon compilation process is automatic.

The two previous algorithms assume serial decoding. If it is desired to do the decoding faster, parallel decoding should be considered. MacPitts has a mechanism for this implicit in the integer data types (which look at a data word in parallel), and the multiple PROCESS algorithm, which performs independent functions in parallel. Parallel data processing will be considered in Chapter VI.

The alternate solution (control path logic) to the Gray code decoder is shown in Appendix B for comparison to gc.mac and gc2.mac. The script and cif files are included also for comparison.

```
;GRAY CODE to BINARY conversion algorithm
(program gc2  2
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def reset signal input 5)
(def inp port input (6 7))
(def bin port output (8 9))
(def 1Ø  power)
(process grycod  Ø
msbs
    (cond((=Ø inp)(setq bin inp)(go msbs))
         ((= 1 inp)(setq bin inp)(go compl)))
compl
    (cond((=Ø inp)(setq bin (word-not inp))(go compl))
         ((= 1 inp)(setq bin (word-not inp))(go nextbit)))
nextbit
    (cond((=Ø inp)(setq bin inp)(go nextbit))
          ((= 1 inp)(setq bin inp)(go compl)))  ) )
```

Figure 4.16 Gc2.mac

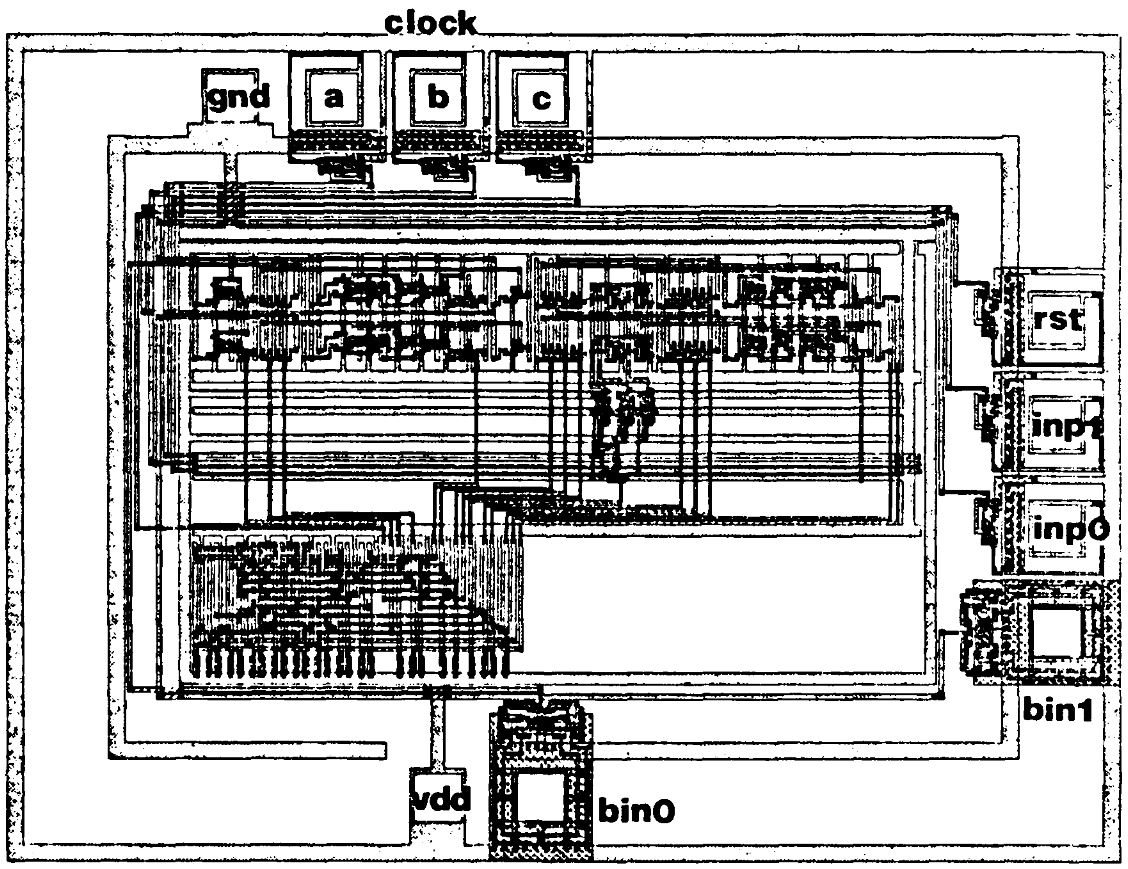


Figure 4.17 Gc2.cif

C. A BLACKJACK GAME

The previous section discussed MacPitts sequential logic implementation as a function of algorithmic syntax. A simple finite state machine was developed, and the structural ramifications of the source algorithm were investigated. This section will discuss development of a more complex algorithm, and its consequent structure.

1. The Algorithm

The blackjack game algorithm was developed based on the following rules. The rules are expressed as FSM states, since the transition to MacPitts syntax is easier that way. The capitalized words correspond to node names and MacPitts variables.

```
s0:START, initialize
s1:ACCEPT card (?) (F, go s0), add FACE value to
   SCORE
s2:if ace and no prior ace valued as 11,
   SCORE=SCORE+10
s3:if SCORE<=16, HIT, go s1
s4:if SCORE>21 and previous ace valued as 11,
   SCORE=SCORE-10, go s5
s5:if SCORE<21 and no previous ace valued as 11,
   BROKE, go s1
s6:if 17<=SCORE=>21, STAND, go s1
```

The next step is to create a state transition diagram, and then to translate the game rules into the appropriate MacPitts entities (ports, registers, signals, and flags).This is usually done from an English description, and then the number of states is minimized by standard techniques. Figure 4.18 shows the transition diagram, which is not minimized for the sake of clarity. There are seven nodes in the diagram. The top node is start, the initial state and the state to which the FSM reverts when the reset signal is brought high. The next node is draw, where the player draws a card (simulated by an off-chip random number generator). The third node is labelled ace, and represents decisions made if an ace is drawn. The next node, htchk, checks for a hit condition (draw another card). Following htchk is devalu, which decrements the rscore contents when appropriate. Then the broke (lose game) condition is tested in the brkchk (broke check) state. Finally, the stand check node, stchk, tests if the stand (win) condition exists, and the program returns to the initial state for either replay or termination. The state transitions follow from the preceeding rules. The MacPitts driver algorithm is written on the basis of the state transition diagram. The driver is shown in Figure 4.19.

Storage elements are required for state transition decisions under the CONDs, so these variables must be flags (aceflg and acptflg). Line 11 in the source code reflects this. The arithmetic comparisons are made on integer values, and these must likewise be storage elements, so this variable is defined as a register (rscore, line 10). Since the FSM progresses asynchronously with the output (no new output with each clock cycle),

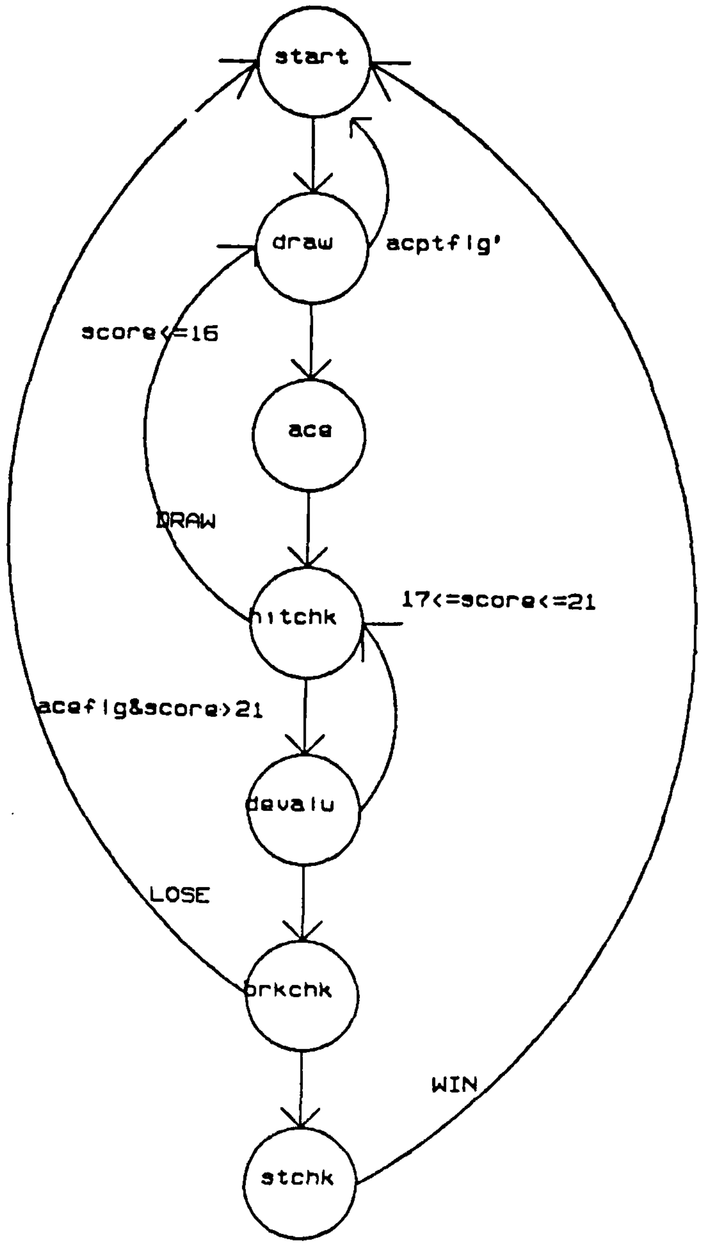


Figure 4.18 Blackjack Game State Transitions

```
;B5.MAC  BLACKJACK MACHINE
(program blackjack  5
(def 1 ground)(def 2 phia)(def 3 phib)(def 4 phic)
(def face port input( 5 6 7 8 9))
(def hit signal output 10)(def stand signal output 11)
(def broke signal output 12)
(def score port output(13 14 15 16 17))
(def accept_card signal input 18)
(def reset signal input 19)
(def 20 power)(def rscore register)
(def aceflg flag)(def acptflg flag)
(always(setq acptflg accept_card))
(process play 0
start
(cond(acptflg(setq rscore 0)(setq aceflg f)))
draw
(cond(acptflg(setq rscore(+ rscore face))
             (setq score rscore)               (go acenode))
    (t                                         (go start)))
acenode
(cond((and (= face 1) (not aceflg))
     (setq rscore (+ rscore 10))
     (setq score rscore)
     (setq aceflg t)))
htchk
(cond((unsigned-<= rscore 16)(setq hit t)      (go draw)))
devalu
(cond((and aceflg (unsigned-> rscore 21) )
     (setq rscore (- rscore 10))
     (setq aceflg f)
     (setq score rscore)
                                               (go htchk)))
brkchk
(cond((and (unsigned-> rscore 21)(not aceflg))
     (setq broke t)                            (go start)))
stchk
(cond((and(unsigned-<= rscore 21)
      (unsigned->= rscore 17))
     (setq stand t)                            (go start)))
                                                        ) )
```

Figure 4.19 B5.mac

there must also be a port (score, line 7) to clock the register value to. Similarly, a port (face, line 4) is defined as the input (face value) of the card. Whenever an output is produced asynchronously with the clock, the latching operation

```
   (setq <register> integer_value)
```

must be made. One method of clocking the register contents to the output port is to use the ALWAYS statement under the PROGRAM statement.

```
   (program <name> <data path width>
      .
      .
   (always(setq(output_port  register_contents)))
   (process <name> <stack depth>
      .
      .
```

This will insure accurate current output values. In the blackjack algorithm, this procedure will not work. If the statement

```
   (always(setq score rscore))
```

is used, the algorithm would appear to work in the command interpreter. Upon compilation, however, the following LISP compiler (Liszt) diagnostic results,

```
  Error: Non-number to minus nil
  <1>
```

where the first line of the diagnostic indicates an attempted arithmetic operation on an empty LISP atom or list, and the second line is the LISP debugger prompt [Ref. 11:p. 11-1].

The reason why this does not work (for this algorithm) is that rscore has not been initialized (as in Fortran, for example) at execution of the ALWAYS statement. The LISP primitive representing rscore is at this time a nil, or empty, atom. The solution is to clock the register (rscore) to the output port (score) whenever it changes value. Lines 18, 21, and 23 show this other method of register transfer to ports.

There are some new forms in b5.mac which also require discussion. The integer test which returns a Boolean value to control is

```
(<signed> <inequality type> integer1 integer2)
```

where the field <signed> is required, and is either blank or the string "unsigned-" for the less than, less than or equal, greater than, or greater than or equal tests. The comparison is made with the <inequality type> between integer1 and integer2.

For instance, if temp is an integer variable set equal to 72, hot is an integer variable set to 88, and cold is an integer variable set to 60, the following forms would produce the signals to control shown. The result of the

FORM	SIGNAL TO CONTROL
(cond((=hot 88)))	T
(cond((unsigned-< hot 99)))	T
(cond((unsigned-<=hot 89)))	T
(cond((= temp hot)))	F
(cond((unsigned-> temp hot)))	F
(cond((unsigned->= 70 temp)))	F

integer comparison test is a Boolean value, and as suchis used as a conditional under COND, as shown in Figure 4.19.

The remaining forms in the algorithm have been previously explained. The algorithm b5.mac (which required five tries to obtain a successful compilation) follows the FSM state transition diagram with the syntax given. The algorithm has been exhaustively tested (only possible with simple FSMs) in the command interpreter.

2. <u>The Chip</u>

Figure 4.20 shows the cifplot resulting from b5.mac. The appearance is similar to the Gray code decoder layout, with the exception of an added functional block at the top right. This is the flag block, resulting from line 11 in b5.mac. The flag block is both a source and a destination for control signals, as the driver syntax suggests.

The data path is organized in five parallel units, as expected from line 2 in b5.mac. There are seven states in the FSM, so only three of the five instantiated sequencer tails are connected to control (the other two are vestigal, instantiated, yet not used). Since four integer values were used in the comparisons,

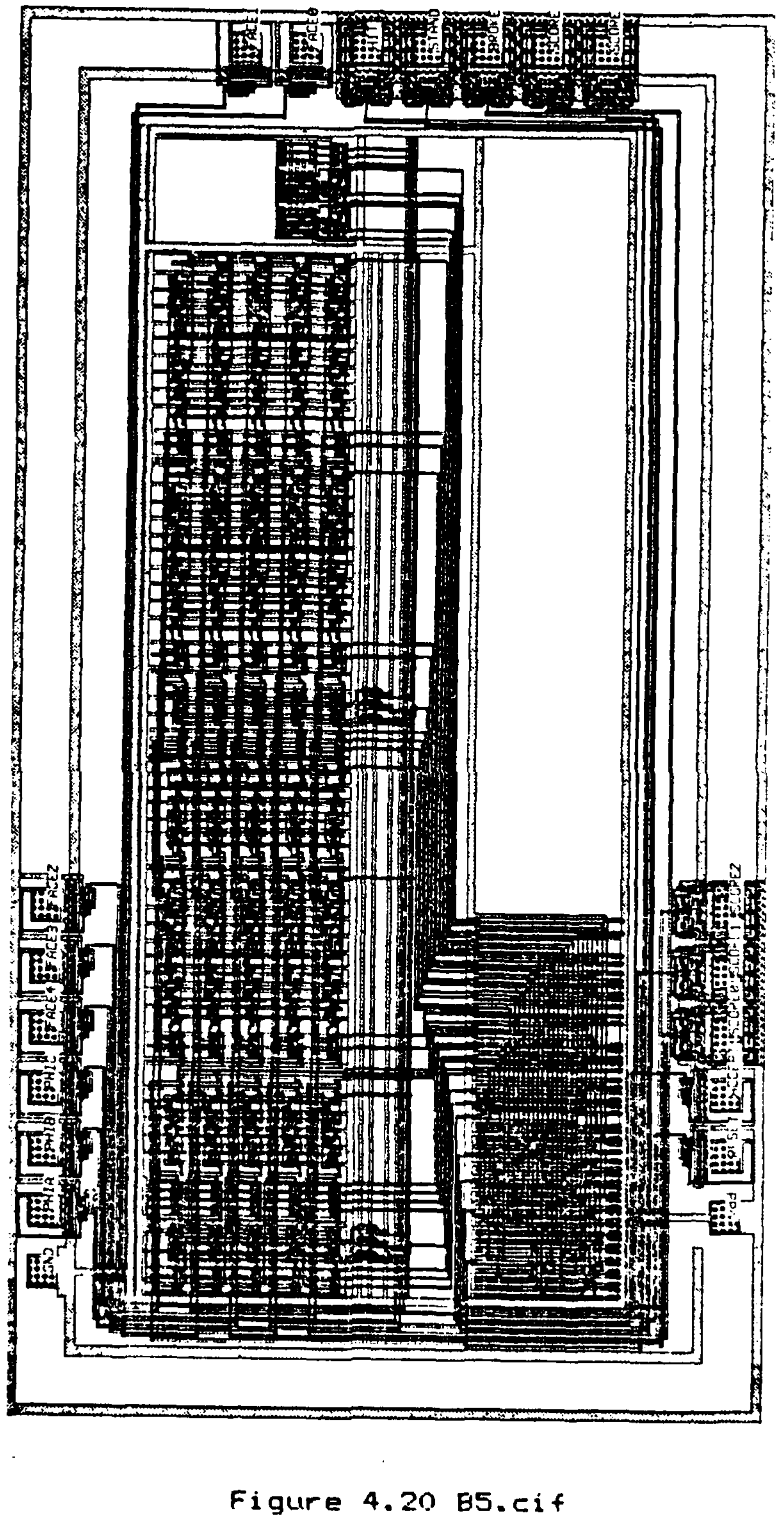


Figure 4.20 B5.cif

the data path is required to generate the comparison integers. This must be considered in designing an algorithm, in assigning the data word length under the PROGRAM statement. The maximum score possible for the blackjack game is 27, so the minimum word width is five. Another reason for the lengthened data path is the number of arithmetic tests made. The integer values for hit, stand, broke, and devalu are made within the data path, since syntax specifies structure in MacPitts. In the Gray code decoder, the comparison tests generate combinational logic in the data path which sends a signal to control. As more data path tests are required, a longer data path will result.

The Weinberger array of the blackjack chip shows a multi-level structure similar to the Weinberger array for the Gray code decoder. As the Weinberger array grows in complexity, it becomes increasingly difficult to understand its function in terms of a gate level equivalent. The correct by construction property of MacPitts is intended to assure correct operation of large control path circuits nonetheless. The compilation session recording in Appendix B shows the MacPitts instantiation process for the blackjack machine, which follows the same general scheme as for the Gray code decoder.

D. MEAD-CONWAY TRAFFIC LIGHT CONTROLLER

The functional description of the Mead-Conway traffic light controller is taken from [Ref.4:p.85]. The chip controls a traffic light at a highway-farm road intersection.

1. The Algorithm

Design of the algorithm follows principles stated previously. After the desired function is understood, an automata (state diagram) is drawn. From this, the algorithm is written. The placement the logic is determined by syntax, and the selection of storage entities (flags or registers) follows.

The light controller controls the three-light traffic signals at the intersection of a busy highway and a less busy farmroad. The input signals are C (car on the farmroad), TL (long timeout), and TS (short timeout). The outputs are ST (start timer), FLO and FL1 (encode the color of the farmroad light), and HLO and HL1 (encode the highway light color). An FSM is appropriate to represent the sequential nature of the traffic light cycling. Figure 4.21 shows the state transition diagram, with labels corresponding to the MacPitts states in the algorithm.

Next, the algorithm is written. A control path architecture is chosen for ease in setting the output bits (initially, the output bits are set individually). Storage elements (flags) are not needed for this example, since

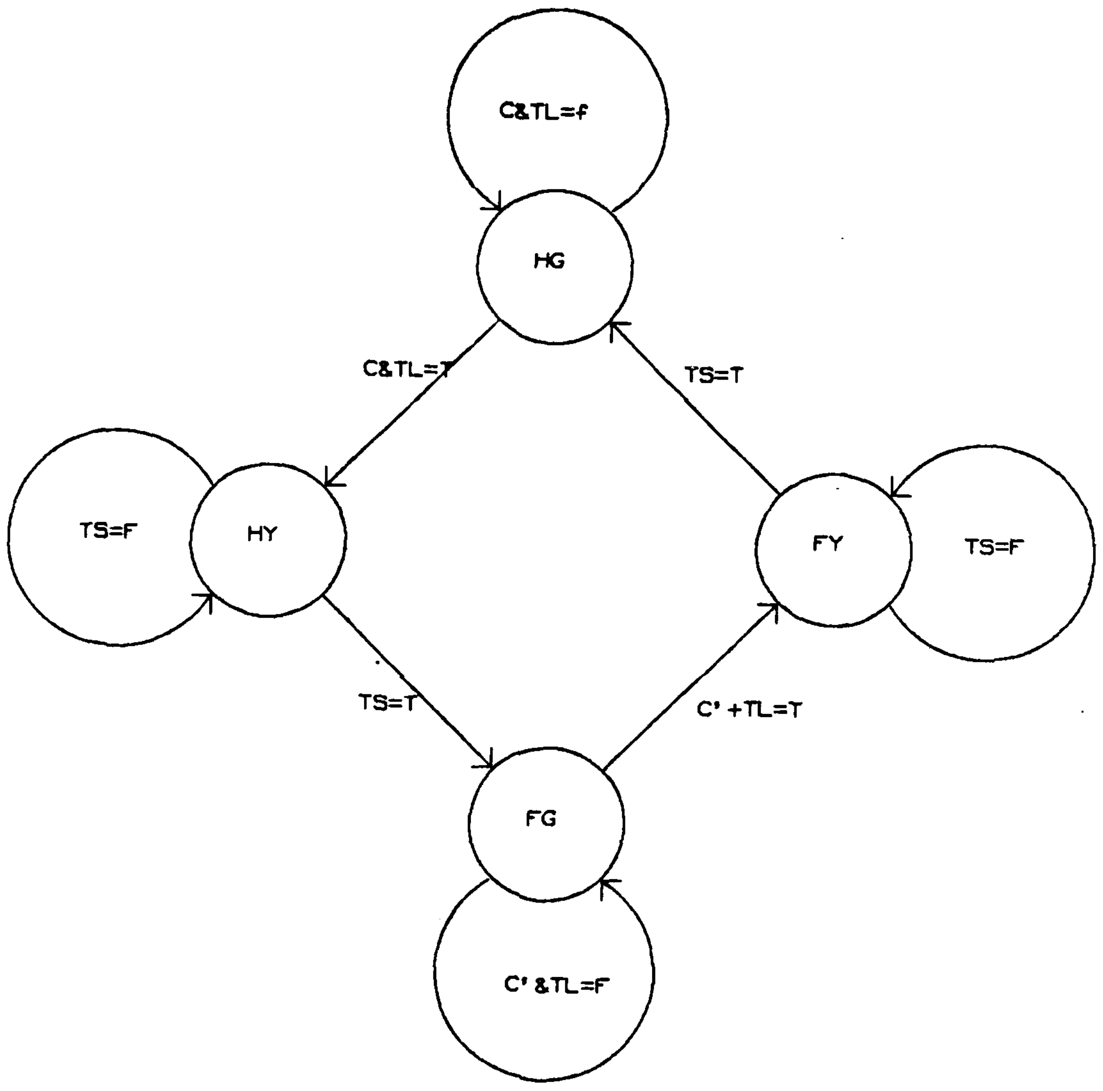


Figure 4.21 Light Controller State Transition Diagram

the outputs are synchronously produced, and constant throughout a given state. In control path circuits using Boolean variables, the value goes to FALSE at the next state transition unless it is explicitly set to TRUE. So storage of the output values would be required if they were to be output within a different state from that in which they are determined. For example, if the light control signals for the highway yellow (HY) state were produced in the previous state (HG), then they would require latching so the correct values would remain after the state transition. If the chip was to be produced, however, the outputs would require latching as explained in the previous section, since the chip clock is many times faster than the light timer clock.

The output bits which control the farmroad and highway light colors must be encoded. The following table is used

HL0	HL1	FL0	FL1	
0	0	0	0	GREEN
0	1	0	1	YELLOW
1	1	1	1	RED

and the output bits are explicitly set to Boolean values in the SETQ forms.

Figure 4.22 is the MacPitts algorithm to create the traffic light controller. The format is similar to the previous FSM drivers, with the exception of absence of data path combinational logic. The data path width must be

```
;MEAD-CONWAY LIGHT CONTROLER

;Set the D.P. width to 2 (4 nodes in FSM dgm)
(program lc2   2
(def 13 power)
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)

;The following 3 SIGNALS are control inputs:
(def c signal input 5)
(def tl signal input 6)
(def ts signal input 7)

;The RESET signal is required for all FSMs:
(def reset signal input 14)

;Define 5 output SIGNALS (=> C. P.) to
;Control the TIMER & HW/FR traffic light:
(def st signal output 8)
(def hlØ signal output 9)
(def hl1 signal output 1Ø)
(def flØ signal output 11)
(def fl signal output 12)
(def fl1 signal output 12)

;The PROCESS statement implies FSM sequencing,
;The stack depth is zero:

(process light_controller  Ø

;The HIGHWAY GREEN state; output=f(PS & PI)
;where <hg>=PS, and <C,TL,TS>=PI:
hg
         (cond((not(and c tl ) )
                                   (setq hlØ f)
                                   (setq hl1 f)
                                   (setq flØ t)
                                   (setq fl1 f)
                                   (setq st  f)
                                          (go hg))
               (t                  (setq hlØ f)
                                   (setq hl1 f)
                                   (setq flØ t)
                                   (setq fl1 f)
                                   (setq st  t)
                                          (go hy)) )

;The HIGHWAY YELLOW state and associated
;outputs & state transitions (<go ----->)
;[see text for output encoding table and
;explanation of state transition syntax] :
hy
         (cond((not ts)
                                   (setq hlØ f)
                                   (setq hl1 t)
                                   (setq flØ t)
```

Figure 4.22 Lc2.mac

```
                                        (setq fl1  f)
                                        (setq st   f)
                                                   (go hy))
            (t                          (setq hlØ  f)
                                        (setq hl1  t)
                                        (setq flØ  t)
                                        (setq fl1  f)
                                        (setq st   t)
                                                   (go fg))    )
;The FARMROAD GREEN state and associated
;outputs & state transitions:
fg
       (cond((not(or tl(not c)))
                                        (setq hlØ  t)
                                        (setq hl1  f)
                                        (setq flØ  f)
                                        (setq fl1  f)
                                        (setq st   f)
                                                   (go fg))
           (t                           (setq hlØ  t)
                                        (setq hl1  f)
                                        (setq flØ  f)
                                        (setq fl1  f)
                                        (setq st   t)
                                                   (go fy))    )
;The FARMROAD YELLOW state:
fy
      (cond((not ts)
                                        (setq hlØ  t)
                                        (setq hl1  f)
                                        (setq flØ  f)
                                        (setq fl1  t)
                                        (setq st   f)
                                                   (go fy))
          (t                            (setq hlØ  t)
                                        (setq hl1  f)
                                        (setq flØ  f)
                                        (setq fl1  t)
                                        (setq st   t)
                                                   (go hg))    ) ))))
```

Figure 4.22 Lc2.mac (continued)

nevertheless declared with the PROGRAM statement. The data path width is two , to permit instantiation of two sequencers to cycle through the four states of the FSM. The initial attempt at lc2 erroneously used a data path width of five , and the algorithm compiled to cif. The resulting cifplot had a data path width of five bits, only two of which were connected to the sequencer tails to remember and address the states. The other three data path units took up chip space, but performed no function.

2. The Chip

The cifplot resulting from lc2.mac is shown in Figure 4.23, and the script of the compilation session is in Appendix B. The cifplot resembles the previous two FSM cifplots, but lacks flags and data path logic. The only registers shown are those which receive and store state information from the sequencer tail. As usual, they lie in the data path above the clock drivers. Other than that, the cifplot for lc2 has no data path. This is expected in view of the driver algorithm, and the script file of the compilation shows only six data path organelles but 43 columns in control. A handcrafted version of this chip could be produced with just a data path, if a two phase clock is used. This will be considered in the next chapter.

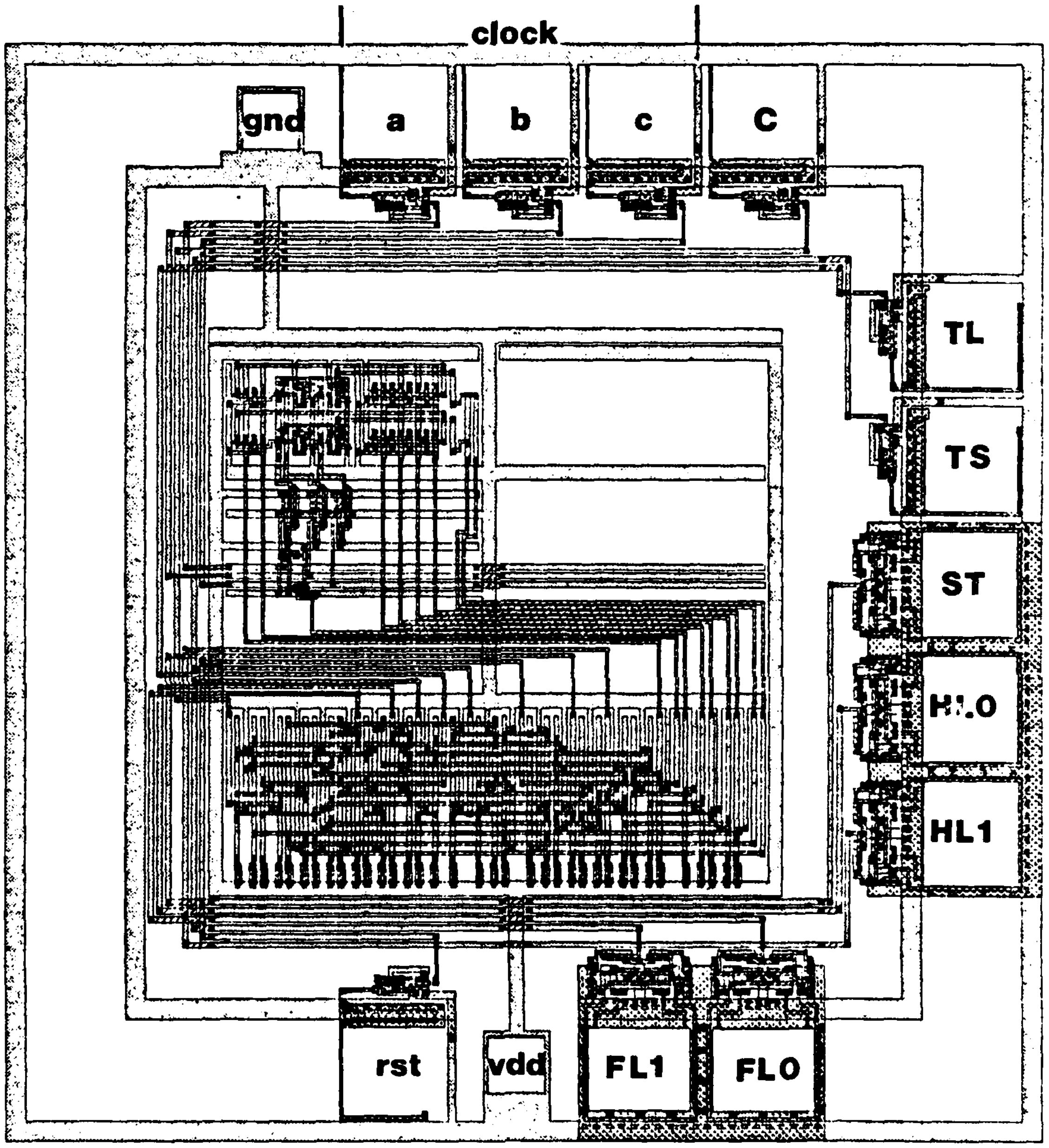


Figure 4.23 Lc2.cif

E. SUMMARY

This Chapter has considered three examples of MacPitts sequential logic: the Gray code decoder, the blackjack game, and the Mead-Conway light controller. In each case, the Mealy FSM convention of MacPitts led to an easy transition from state diagram to algorithmic description. The Mealy architecture is evident in both the MacPitts algorithm and the resulting chip layout.

In the algorithm, each state is given a name (e. g., HIGHWAY GREEN, HIGHWAY YELLOW) and within each state the outputs are determined with the COND form and set accordingly. The output is a function of both present state and present input (e. g., CARS, TIMEOUT_LONG, TIMEOUT_SHORT).

The same Mealy logic is evident in the circuit layout (cifplot). The sequencer stores the present state, and multiplexers driven by the Weinberger array and present inputs determine the next-state transitioning by controlling the inputs to the bank of present-state registers.

Sequential logic in MacPitts can be influenced by the designer in the same way as combinational logic can, by explicitly specifying the desired outputs. The alternative is to specify the outputs as an implicit function of either inputs (ports, input signals) or intermediate results (internal signals, flags, registers). In general, when the explicit specification of outputs is used

```
  (setq score 19)
```

rather than the functional specification of outputs

```
  (setq score (+ rscore face))
```

a smaller and faster circuit will result. The explicit specification of outputs is therefore the preferred method, though not always possible. If there are many possible outputs, it may even be better to use the functional specification of outputs rather than attempting to specify each one explicitly.

The data path width for a MacPitts sequential machine, as specified in the PROGRAM statement, must be large enough to address the number of states. That is, the data path width must be greater than or equal to log (base 2) of the number of states in the state transition diagram. If this condition is not met, MacPitts (the silicon compiler) will not successfully compile the source algorithm. The reason for this requirement is the manner in which MacPitts lays out the sequencer and data path. The sequencer and data path are laid out contiguously, in a linear bit-slice configuration. The width of both is the width of the data path as specified in the PROGRAM statement (this number is also the number of present-state registers instantiated). Since there must be the same number of i/o ports as the data path width, and since all of these ports may not be used for data i/o, one solution to the problem of extra ports is to ground them in the circuit in which the chip is to be used (as suggested for the Gray code decoder, where only one port was necessary, but two ports had to be specified to allow enough state transitions). The alternate solution for the Gray code to binary conversion routine is to treat the data as a serial stream, one bit wide. This suggests using SIGNALs (instead of PORTs) as inputs, and processing the Gray code as Boolean data instead of integer data. This algorithm is included for completeness in Appendix B, with the resulting cifplot and script of the compilation process.

MacPitts provides a convenient method to compare both Boolean and integer values, which is particularly useful in the decision-making under a COND. The Boolean comparisons (Figure 4.22) are used to test the value of a flag or a signal, and the integer comparisons (Figure 4.19) are used to compare numerical values in ports or registers. In each case, the result is a Boolean signal to control which affects subsequent state transitioning or setting of outputs.

Algorithm design for MacPitts FSMs begins with the decision of how much data it is desired to process simultaneously, and in what form that data presents itself to the chip. For instance, if a serial FSM chip is desired (e. g., a serial Gray code decoder), the data word is one bit wide. The inclination is therefore to treat the data as Boolean type, which is feasible for FSM architectures for reasons explained previously. The designer is not constrained to integer data types in this case (although the examples presented in Figure 4.2 and Figure 4.16 used integer data types). If the data comes to the chip for parallel processing in an n-bit word, however, the inclination is to treat the data as integer type (for example, the blackjack algorithm). This is not always possible, for reasons to be explained in connection with Hamming error correction in Chapter VI (MacPitts does not permit implicit setting of bits within a data word).

Algorithm design may be viewed as the designer's influencing of the chip layout. Since circuit structure is a function of syntax (on a lower level), it is reasonable to assume that chip layout is a function of algorithm structure (on a higher level). That is, syntax determines not only the individual circuit elements (NANDs, ORs, XORs, ports, flags, registers, etc.) of the chip, but also determines how the individual elements work in concert. The source algorithm lc2.mac shown in Figure 4.22 used Boolean control signals as inputs (C, TL, TS). The resulting cifplot in Figure 4.23 shows a Weinberger array at the bottom, and no data path except for a bank of two sequencer organelles at the top of the chip. This chip can be viewed as a control path chip. An alternate design would use a five-bit word (representing the signals HL0, HL1, FL0, FL1, and ST) as the output, and retain the three control signals as inputs. Appendix B shows dplc2.mac (the data path equivalent of Figure 4.21, lc2.mac), and the resulting cifplot. The output bits are set explicitly by setting the output word values in the .mac file. This results in a larger data path, as expected, since the output decisions result in data path operations instead of control path operations. The control path is smaller than in lc2.cif, since the Weinberger array has fewer decisions to make. Appendix B also contains the script file of the compilation of dplc2.mac.

Yet another version of the light controller would assign the input values to a three bit word (representing C, TS, and TL), and make the conditional checks on the input control word with the BIT statement. This solution would result in a still larger data path and a smaller control path than the two previous light controller chips. Just as in any high-level language, there exists many ways of solving a given problem with MacPitts. The best way to solve the problem must consider not only the algorithm, but the structural (layout) consequences of algorithmic syntax. The "best" solution is arrived at by experience in MacPitts programming, knowledge of the consequences of syntax, and finally, iteration toward a better solution (trial and error).

V. A COMPARISON OF A MACPITTS DESIGN WITH A HANDCRAFTED EQUIVALENT

Previous chapters illustrated some inefficiencies inherent in the MacPitts layout scheme. The Weinberger array and the data path both use transverse polysilicon wires for cross-communication, and poly has the highest specific resistance of the three possible NMOS wire materials. The one dimensional river routing method used is not optimal, because the input, output, and data/control lines required are long. The sequencer organelles are instantiated according to the data path width, and not according to the number of states necessary. The Weinberger array generates multiple cascaded gates to implement multiple output combinational logic functions, causing long signal delays in comparison to a PLA. A handcrafted version of a functionally equivalent chip is compared to a MacPitts design to investigate these differences both quantitatively and qualitatively.

A. THE HANDCRAFTED TRAFFIC LIGHT CONTROLLER

The standard for this comparison is a handcrafted (CAD) version of the Mead-Conway traffic light controller which is compared to the MacPitts generated version in terms of speed and power consumption. Qualitative observations are also described.

The custom-made traffic light controller was constructed on the Caesar VLSI graphics editor with the aid of various VLSI CAD tools.

1. Design

The MacPitts-produced traffic light controller was described in the Chapter IV. MacPitts design is just a matter of generating a prototype MacPitts driver program, and refining it until an acceptable archetype algorithm is achieved. This is done in both the command interpreter (algorithmic optimization), and in Caesar (structural optimization). Caesar allows the designer to see the structure and analyze it with power estimators (Powest) and timing estimators (Crystal, SPICE). Moving pads and deleting vestigal structures are examples of possible structural optimizations using Caesar (this procedure should be considered if the MacPitts chip is to be fabricated).

The standard VLSI design scheme is similar to MacPitts design in that structure is considered as a function of behavior. The behavior is not constrained to follow a given algorithmic syntax, though, as it is in MacPitts. So custom design is more flexible than silicon compiler designs are, since the designer can choose any desired structure to implement the behavior called for.

The standard NMOS PLA is used for the hand-crafted traffic light controller. Mead and Conway [Ref. 4:pp.80-88] develop the state transition table for the light controller.

and provide a sticks diagram of the clocked PLA FSM. The following PLA is based on the Mead-Conway development.

Ousterhaut [Ref. 9] illustrates use of Eqntott and Reference 10 illustrates use of Tpla to generate this PLA. Eqntott is a VLSI CAD program which takes logic equations as the input and produces a PLA truth table as the output. This truth table is the input to Tpla (Technology independent Programmed Logic Array), and Tpla further allows the designer to geometrically modify the PLA. The result of Tpla processing the truth table is a Caesar representation of the desired PLA. Figure 5.1 shows the input logic equations for Eqntott, and Figure 5.2 shows the resulting truth table from Eqntott.

The best method to design a PLA is to create the logic equations as in Figure 5.1, and then use the Unix pipeline to send the result of Eqntott to Tpla

```
eqntott [options] infilename ! tpla [options]
outfilename
```

The result is a Caesar file of the PLA layout, which must be converted to cif in Caesar as previously described. Figure 5.3 shows the -trans PLA (inputs and outputs on opposite sides) generated from the command

```
eqntott -l -f -R stoplt!tpla -s Btrans -l -O -o
stoplt.ca
```

which took 28 seconds to complete. The eqntott switch -l

```
INORDER =   c tl ts yØ yl;
OUTORDER =  yØ yl st hlØ hll flØ fll;
flØ  =  !c & !yØ & !yl;
yØ   =!(!c & !yØ & !yl);
yl   =!(!c & !yØ & !yl);
flØ  =  !tl & !yØ & !yl;
yØ   =!(!tl & !yØ & !yl);
yl   =!(!tl & !yØ & !yl);
st   =  c & tl & !yØ & !yl;
flØ  =  c & tl & !yØ & !yl;
yØ   =!(c & tl & !yØ & !yl);
yl   =  c & tl & !yØ & !yl;
hll  =  !ts & !yØ & yl;
flØ  =  !ts & !yØ & yl;
yØ   =!(!ts & !yØ & yl);
yl   =  !ts & !yØ & yl;
st   =   ts & !yØ & yl;
hll  =   ts & !yØ & yl;
flØ  =   ts & !yØ & yl;
yØ   =   ts & !yØ & yl;
yl   =   ts & !yØ & yl;
hlØ  =   c & !tl & yØ & yl;
yØ   =   c & !tl & yØ & yl;
yl   =   c & !tl & yØ & yl;
st   =   !c & yØ & yl;
hlØ  =   !c & yØ & yl;
yØ   =   !c & yØ & yl;
yl   = !(!c & yØ & yl);
st   =   tl & yØ & yl;
hlØ  =   tl & yØ & yl;
yØ   =   tl & yØ & yl;
yl   = !(tl & yØ & yl);
hlØ  =   !ts & yØ & !yl;
fll  =   !ts & yØ & !yl;
yØ   =   !ts & yØ & !yl;
yl   = !(!ts & yØ & !yl);
st   =    ts & yØ & !yl;
hlØ  =    ts & yØ & !yl;
fll  =    ts & yØ & !yl;
yØ   =  !(ts & yØ & !yl);
vl   =  !(ts & yØ & !yl);
```

Figure 5.1 Stoplight Logic Equations for Eqntott

```
.i 5
.o 7
.na stoplt
.ilb  c tl ts yØ yl
.ob  yØ yl st hlØ hll flØ fll
.p 12
.f 1 4
.f 2 5
Ø--ØØ  Ø Ø Ø Ø Ø 1 Ø
-----  1 1 Ø Ø Ø Ø Ø
Ø----  1 Ø Ø Ø Ø Ø Ø
-Ø-ØØ  Ø Ø Ø Ø Ø 1 Ø
--ØØ1  Ø 1 Ø Ø 1 1 Ø
--Ø1Ø  1 Ø Ø 1 Ø Ø 1
Ø--11  1 Ø 1 1 Ø Ø Ø
--1Ø1  1 1 1 Ø 1 1 Ø
--11Ø  Ø Ø 1 1 Ø Ø 1
-1-11  1 Ø 1 1 Ø Ø Ø
1Ø-11  1 1 Ø 1 Ø Ø Ø
11-ØØ  Ø 1 1 Ø Ø 1 Ø
.e
```

Figure 5.2 Truth Table Input for Tpla

means list the truth table, -f means to connect the feedback paths in the PLA, and -R directs eqntott to minimize the truth table. The tpla switch -s selects the PLA type (-trans shown), and -I and -O indicate clocked inputs and outputs. This command string creates an NMOS FSM Caesar file. It was determined later that a -cis PLA (input and output on the same side of the PLA) would fit the chip frame better. The change is simple. The same command string as above was issued, except Bcis was substituted for Btrans.

The PLA is a fast structure. Appendix A shows the interactive Crystal session showing the timing analysis of just the PLA. The delays are determined to be 26.93 ns for phia and 32.06 ns for phib. For symmetric phia and phib durations, with each having the duration of the slowest critical path, or 32.06 ns, the maximum clock rate is 15.6 ns. The maximum clock rate is calculated as the inverse of twice the slowest critical path time. The use of Crystal on non-overlapping, two-phase clocking schemes is described in [Ref.3:pp. 80-93].

The sequential logic for the light controller chip is made with the University of Washington/Northwest Consortium CAD tools as described above. All that is lacking is the power and ground connections and the pads. Usually the power and ground busses are laid out by hand (Caesar) or specified in cartesian coordinates (CLL, Chip Layout Language, a method of specifying mask polygons, their

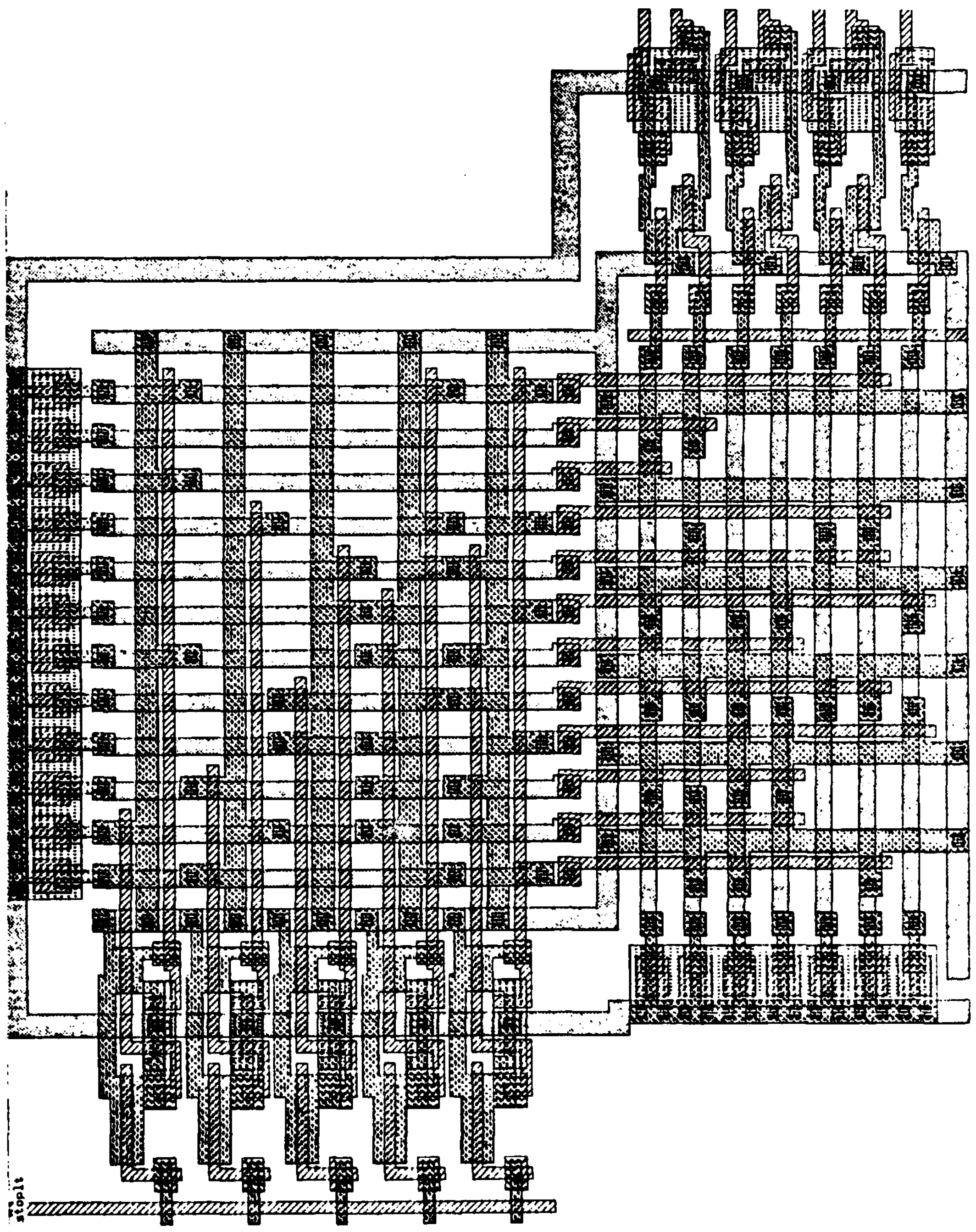


Figure 5.3 -Trans PLA Resulting from Eqntott and Tpla

dimensions, and the fabrication process required), and the pads are then invoked from an existing library of VLSI macro cells. MacPitts can shorten the design time by doing most of this work for the designer. Figure 5.4 shows the algorithm stoplt_frm.mac used to create the frame for the PLA FSM. The frame is created like wire.mac (Figure 2.1), in that it is just wires from input to output. The wires are deleted in Caesar, and the PLA is placed in the center of the chip frame. Figure 5.5 shows the resulting chip. The clocked -cis PLA is in the center of the chip, connected to appropriate inputs and outputs (tpla makes this connection easy, it labels all inputs and outputs). The third clock pad (phic) is deleted in Caesar. This chip still has long indirect metal runs and lots of white space.

2. Optimization and Analysis

Figure 5.6 shows a condensed version of the chip, stoplt_minc.cif. The area of the chip shown in Figure 5.6 is 40% smaller than the chip in Figure 5.5, and still more reduction is possible. Since there are 12 pads, it would be better to place three per edge on the chip. The signal wires could also be shortened by judicious choice of pad placement in the .mac algorithm. And finally, all sides could be brought closer together. There exists a synergistic relationship between the existing CAD tools and MacPitts that bears further study.

```
;stoplt_frm.mac
;This pgm creates a design frame for the stoplight
;controller [cf.Mead & Conway, p.81, 2nd printing]
;hand-crafting will be required to merge the PLA
;FSM created by eqntottitpla into this frame. CAESAR
;is used to do this.
(program stoplt_frm.mac 5
(def 13 power)
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
;inputs to light controller PLA FSM
(def  c signal input 5)
(def  tl signal input 6)
(def  ts signal input 7)
;outputs from light controller PLA FSM
(def  st signal output 8)
(def  hl0 signal output 9)
(def  hl1 signal output 10)
(def  fl0 signal output 11)
(def  fl1 signal output 12)
(always
;
;here we setq 5 simple dummy paths. These are chosen with a
;view towards later simple editing in CAESAR
;
(setq st c)
(setq hl0 tl)
(setq hl1 ts)
(setq fl0 c)
(setq fl1 tl)  )     )
```

Figure 5.4 Stoplt_frm.mac

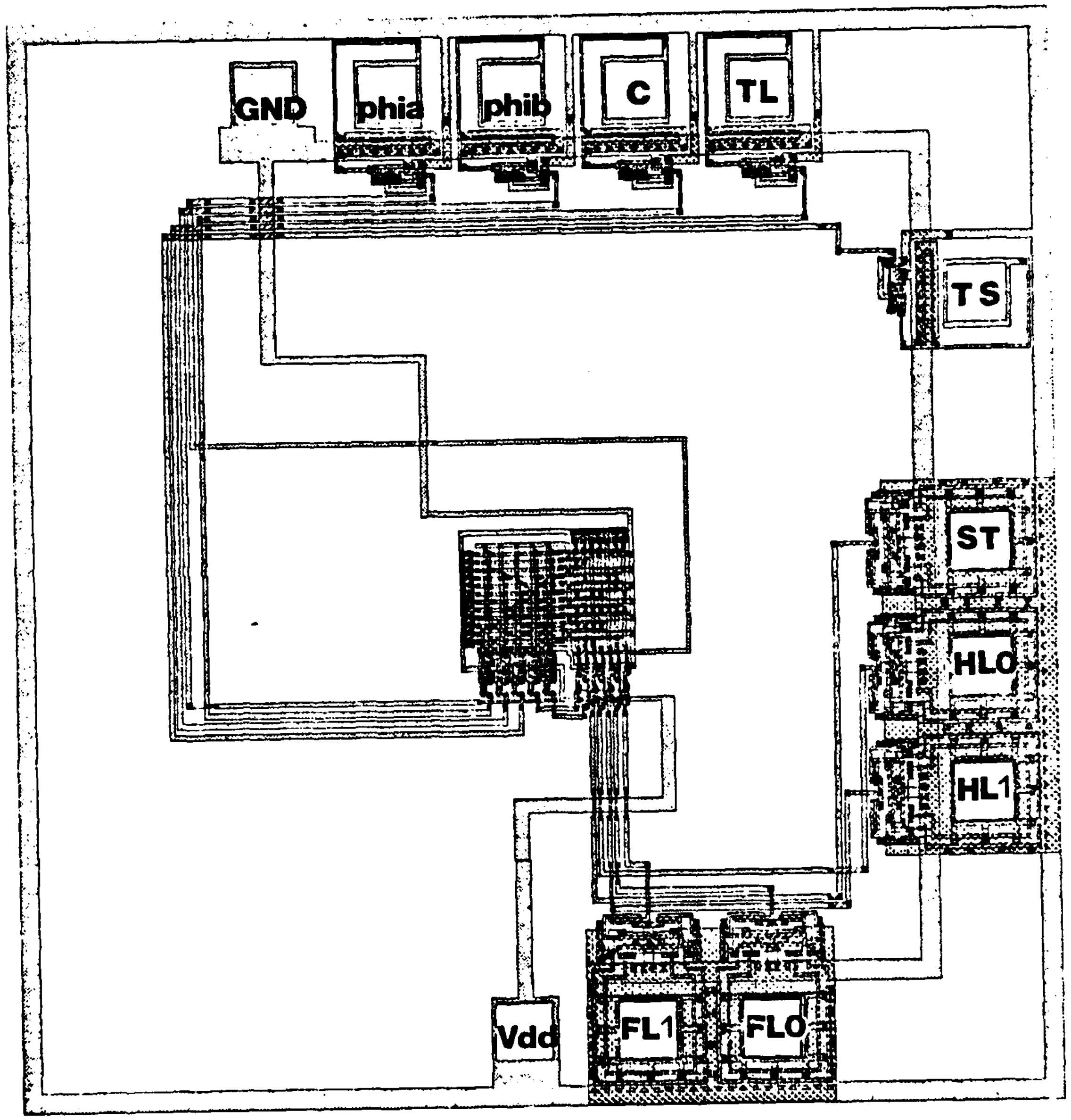


Figure 5.5 Stoplt_chp.cif

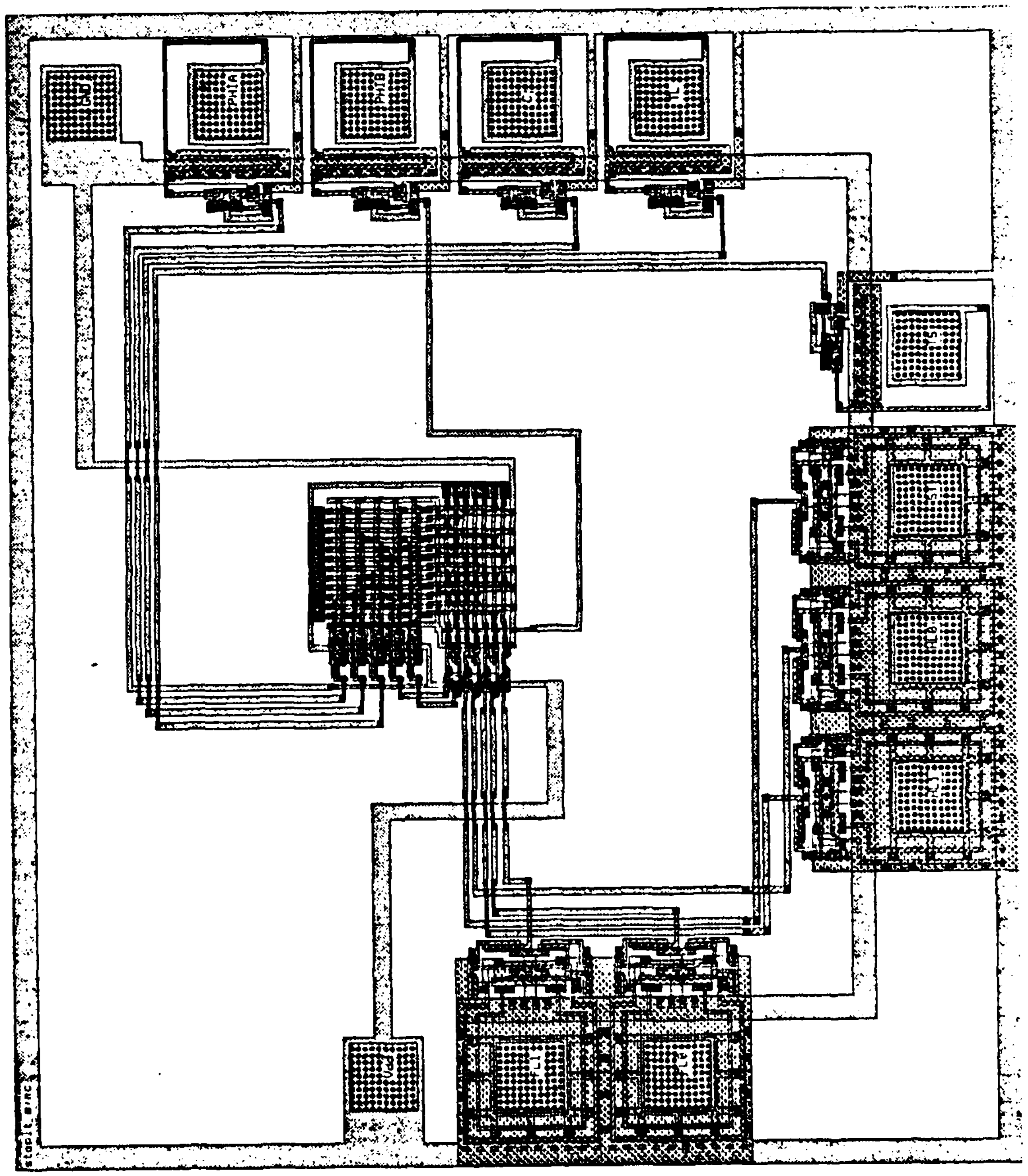


Figure 5.6 Stoplt_minc.cif

Intervention by the designer, however, is antithetical to the goal of silicon compilation. The silicon compiler has a ruleset which (in theory) guarantees the property of "correct by construction". This property states that the chip design will always be functionally correct; it cannot be wrong. Circuit density is not the primary goal, nor is speed.

The MacPitts designer has no control over circuit density, other than Boolean optimization of the algorithmic forms as explained in Chapters II and III. The designer does have some control over chip speed. There are two ways of optimizing throughput in a MacPitts design. The first method is explained at the beginning of Chapter III, and can be thought of as algorithmic optimization. The objective is to write an algorithm which executes in a minimum number of clock cycles. The verification is done in the command interpreter. PAR, COND, and PROCESS are used wherever possible to parallel operations.

The second method of controlling chip speed is through circuit optimization (this too is a function of syntax in MacPitts). The designer chooses either the data path or the control path or a hybrid of both, and with Crvstal designs a chip which has a maximum speed per clock cycle. The throughput is then the product of the inverse of the number of clock cycles required for a valid result and the cycle rate (results/cycle x Hz = results/sec).

Furthermore, the circuit speed can be increased by judicious placement of pads in the .mac file. It is not always apparent where the routing will go beforehand, so the recommended method is to create a prototype cifplot, and then modify the pad numbering in the .mac file to decrease signal path lengths from the pads to the logic elements. For example, in stoplt_minc.cif (Figure 5.6), the phia pad would be moved to center left on the chip frame, phib to center right, ground to top right, and C, TL, and TS would be moved to the lower left corner region to decrease metal run lengths. All of these suggested moves are not possible due to the way MacPitts places pads, so Caesar editing is required to optimize the MacPitts design if minimal length runs are desired.

Appendix C contains the Crystal analysis of the PLA traffic light controller. The chip speed is limited to the inverse of the sum of the critical propagation times, or 6.85 MHz. This is less than half the speed of just the PLA (16.95 MHz). Appendix C also contains the Powest analysis of the PLA traffic light controller.

B. COMPARISON WITH MACPITTS DESIGN

Appendix C contains the Crystal command file for the MacPitts traffic light controller timing analysis. Froede [Ref. 3:pp. 80-85] explains the analysis of a MacPitts design with Crystal. The Crystal command file in Appendix C shows just the commands issued to Crystal, and in parentheses to the right, the time delay values returned (representing an actual Crystal session).

Figure 4.23 shows the chip on which this Crystal analysis was done. The critical path is from phic to the clock drivers to the state registers. The clock drivers induce a cumulative delay of 23.9 ns, and the state registers a cumulative delay of 114.2 ns, so the transition induces a delay of 90.3 ns. The Weinberger array induces another 178 ns, and the slowest path is from there to the ST pad. The total delay is 363.52 ns, for a maximum speed of 2.75 Mhz. This speed is 40% of the maximum speed of the PLA light controller.

Figures 5.7 and 5.8 show the floorplans of each version of the traffic light controller. Figure 5.7, the PLA FSM is comparatively simple. The FSM is a small clocked PLA with feedback. The connections to the pads are all metal (not shown). Figure 5.8 is the MacPitts version, and is far more complicated. The control path is large, and induces the largest part of the delay. The present state (PS in Figure 5.8) -next state mechanism is much more complex than the simple PLA feedback generated by eqntott and tpla. The wires between the data and control paths are poly, as are the PS feedback lines in Figure 5.8. These wires contribute to the slowness of the MacPitts chip. The wires to the pads also take a more circuitous route, inducing still more delay.

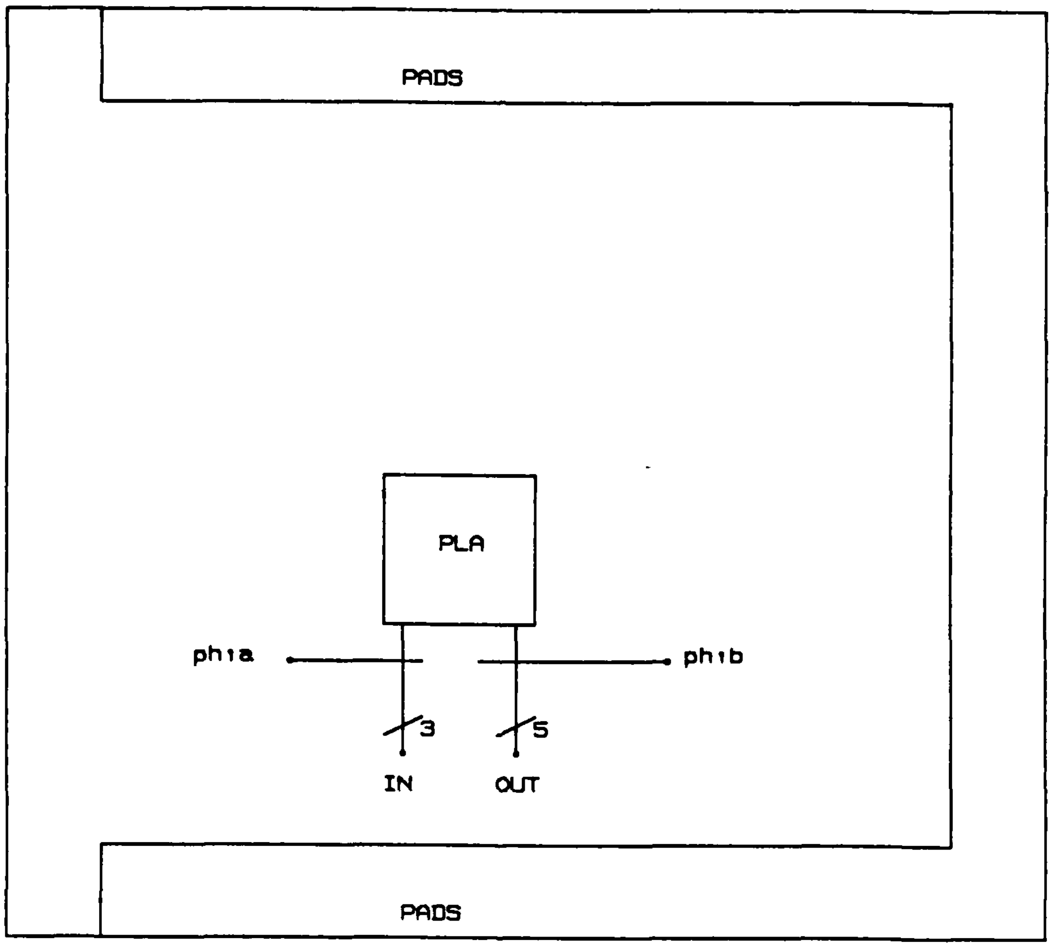


Figure 5.7 PLA Stoplight Chip Floorplan

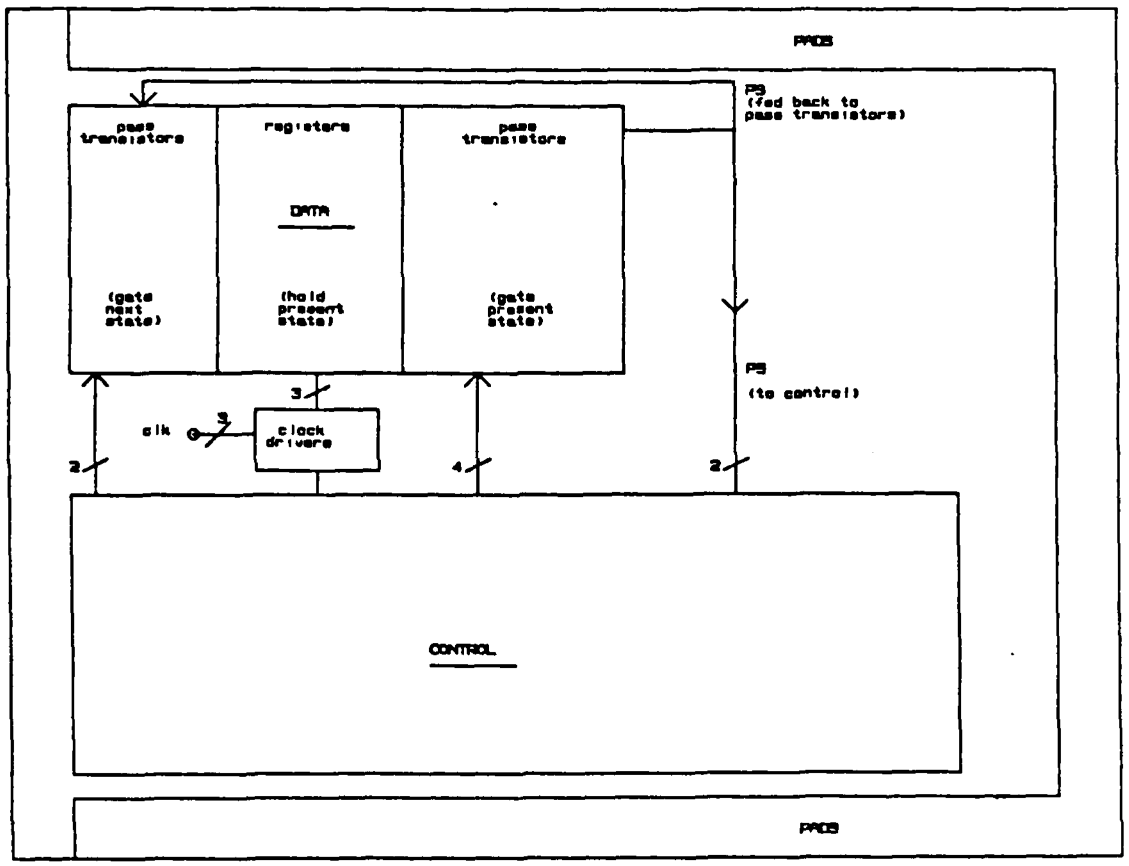


Figure 5.8 MacPitts Stoplight Chip Floorplan

Table 5.1 compares the MacPitts traffic light controller and the PLA traffic light controller.

TABLE 5.1

MACPITTS vs. HANDCRAFTING

	PLA Chip	MacPitts Chip
delay [ns]	146.98	501.97
max. clock freq. [Mhz]	6.85	1.99
pullup transistors	35	87
avg. DC power[W]	.042	.055
max. DC power[W]	.085	.107
control path dimensions [mm]	.49 x .29	.547 x .185
data path dimensions [mm]	.178 x .173	.256 x .240
area ratio [cp/dp]	.046	8.9
chip size [mm**2]	.836	1.164

VI. DESIGN EXAMPLE:HAMMING ERROR DETECTOR/CORRECTOR

This Chapter describes one method of design with MacPitts. The procedure is to first define the problem, then to write an initial algorithmic description of the solution in MacPitts (the language). The initial algorithm is either a simplified version, or a piece of the larger problem. The simplified algorithm is tested for execution in the interpreter, and then compiled to cif. Alternate solutions are considered next, and simplified alternate solutions are likewise tested. The best of these algorithms is then chosen, based on speed, power dissipation, and size. The chosen solution is then expanded to solve the larger problem.

The problem is to design a parallel Hamming method error detector/corrector which will correct single bit errors in a 15-bit encoded message.

A. THE ERROR DETECTOR

The theory behind Hamming error detection and correction is found in most texts on coding and information theory [Ref. 5:pp. 39-49]. A subset of this problem is error detection, which the prototype algorithm solves.

The prototype algorithm looks at a three bit encoded message in parallel, and by the Hamming method determines the bit error location. The algorithm is written to demonstrate correct operation for three-bit messages. It can later be expanded to cover longer word lengths.

The Hamming method scans the encoded word, and by a series of parity checks determines the bit error position. The single error detection method assigns the result of each parity check to a bit of data. The word formed from the resulting bits comprises the syndrome. The value of the syndrome is the bit error position in the received message.

The parity checking is done in a specific order. If the codeword is a string of n bits with the lsb leading

0 1 2 3 4 5 6 7 8 ... n

then the syndrome bits are determined by parity checks across the message bits as shown below.

syndrome bit	message bit positions for parity check
0	0 2 4 6 8 10 12 14 16 18 20 ...
1	1 2 5 6 9 10 13 14 17 18 21 22 ...
2	3 4 5 6 11 12 13 14 19 20 21 22 ...
3	7 8 9 10 11 12 13 14 23 24 25 26 27...
.	.
.	.
.	.

Where the syndrome word is read from msb to lsb and points to the message bit which needs correcting.

For instance, for an encoded seven bit message, there are three check bits (represented by "c"), and four bits of information (represented by "i") in the positions indicated below

```
0 1 2 3 4 5 6
c c i c i i i
```

The first bit of the syndrome (lsb) is determined by parity checks over positions 0, 2, 4, and 6. The next bit of the syndrome considers positions 1, 2, 5, and 6. The last bit of the syndrome (msb) is determined from message positions 3, 4, 5, and 6. The three-bit syndrome indicates the error position in the message string. If the received message is 0100011, the syndrome generated is 011. The syndrome indicates an error in the third bit from the right. The correct message is 0110011. The Hamming method corrects (complements) the third symbol.

1. Design Considerations

Previously in this research it was noted that MacPitts syntax does not permit explicit bit manipulation in the data path. To do this algorithm in the data path may be desirable, in view of the speed of simple data path functions. Since this is not possible, perhaps a hybrid data path-control path algorithm should be considered. A review of the Gray code decoder chip (Figure 4.2) will show why this is not a good approach. The Gray code decoder is a mixed structure, having both a data path and a control path. The interconnections are all poly, which slows the chip down. The multiple unPARalelled CONDs have a more detrimental effect on speed, since each require a clock cycle to execute if its antecedant is true. So the target architecture will be Boolean (control path).

The parity checks can be done by a variety of methods in MacPitts. The simplest way is with the built in library function PARITY, which has the format

```
  parity (boolean boolean ...)
```

PARITY performs modulo two addition, and returns Boolean TRUE to control if the argument is an odd number of TRUEs, or Boolean FALSE if the argument is an even number of TRUEs. So the parity checks can be done directly on the bits of the message, in parallel, with the PARITY statement.

MacPitts also has a method of checking specific bits in a data word. The BIT statement looks at a bit in the integer-valued word, and returns a TRUE to control if the bit is one, or a FALSE to control if the bit is zero. The form of the BIT statement is

```
  (bit  <bit_position> <integer_ex -ession> )
```

Figure 6.1 is the algorithm tst.mac, used to test the BIT statement. It is similar functionally to wire.mac, in that it sets an output bit to an input bit. The difference is that BIT permits a bit-by-bit conversion from integer value to Boolean value. In Figure 6.1, the input word mesg is integer valued. The output bits are Boolean signals (outx), and they are setq'd to the respective bit position values of mesg (the corrupted input word).

2. Prototype Error Detector

Knowing Hamming error detection theory and the PARITY and BIT statement syntax, an error detector algorithm

```
;TST.MAC
;A MacPitts algorithm for bit-setting of output ports
;The BIT form is used to select a specific bit of the
;Input data word, and an output signal is set to
;The value of the bit selected.

;Require a D.P. width of three to accommodate the input:
(program tst    3

(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)

;Use a 3-bit INTEGER as input PORT:
(def mesg port input (5 6 7 ))

;Use 3 BOOLEAN SIGNALS as outputs:
(def outØ signal output 8)
(def out1 signal output 9)
(def out2 signal output 1Ø)
(def 11 power)

;Perform bit-setting on each clock cycle:
(always

;Select which bit of the input word is to
;Be SETQ'd to the output signal pads:
(setq outØ (bit Ø mesg))
(setq out1 (bit 1 mesg))
(setq out2 (bit 2 mesg))  )  )
```

Figure 6.1 Tst.mac

can be written. The encoded message input (mesg) is word-valued, three bits wide. The output syndrome (syndx) is two Boolean signals. The algorithm is shown in Figure 6.2. The semantics of the MacPitts algorithm follow the English description of the problem statement. The appropriate bit patterns of the message are checked, and the syndrome bits are set based on the results of the parity checks. This algorithm was exhaustively tested in the command interpreter, and serves as the prototype for the error

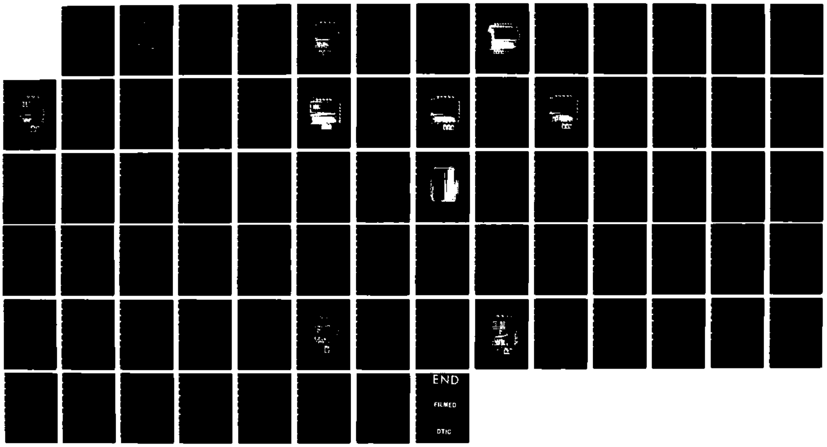
```
;HAM3.MAC
;A MacPitts algorithm for single-error detection
;using the Hamming method.
(program haml 3  ;note width of data path (=width of msg)
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
;mesg is the input data word of 3 bits width with possible errors
(def mesg port input (5 6 7 ))
(def syndl signal output 8)
(def synd2 signal output 9)
(def 1Ø power)
(always

;For a 3 bit word, two parity checks are required. The
;result of these parity checks is a 2 bit syndrome, which
;indicates the bit position of the error in the 3 bit word.

;this cond sets or clears the lsb of the syndrome.
(cond
     ((parity (bit Ø mesg) (bit 2 mesg) )
     (setq syndl t ))
     (t
     (setq syndl f )))

;This cond sets or clears the msb of the syndrome.
(cond((parity (bit 1 mesg) (bit 2 mesg) )
     (setq synd2 t ))
     (t
     (setq synd2 f )) )  )    )
```

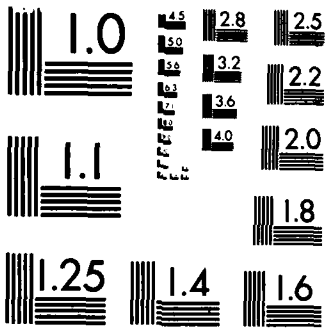
Figure 6.2 Ham3.mac

detector. The algorithm compiled to cif, and Figure 6.3 shows a logic structure completely in the control path. The parallel lines at center left are the input (mesg) bits, and result from the BIT statement. They go to the right side of the Weinberger array, where they fan out to multiple NOR gate inputs.

AD-A161 098 VLSI (VERY LARGE SCALE INTEGRATED CIRCUITS) DESIGN WITH THE MACPITTS SILICON COMPILER(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R C LARRABEE SEP 85 3/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

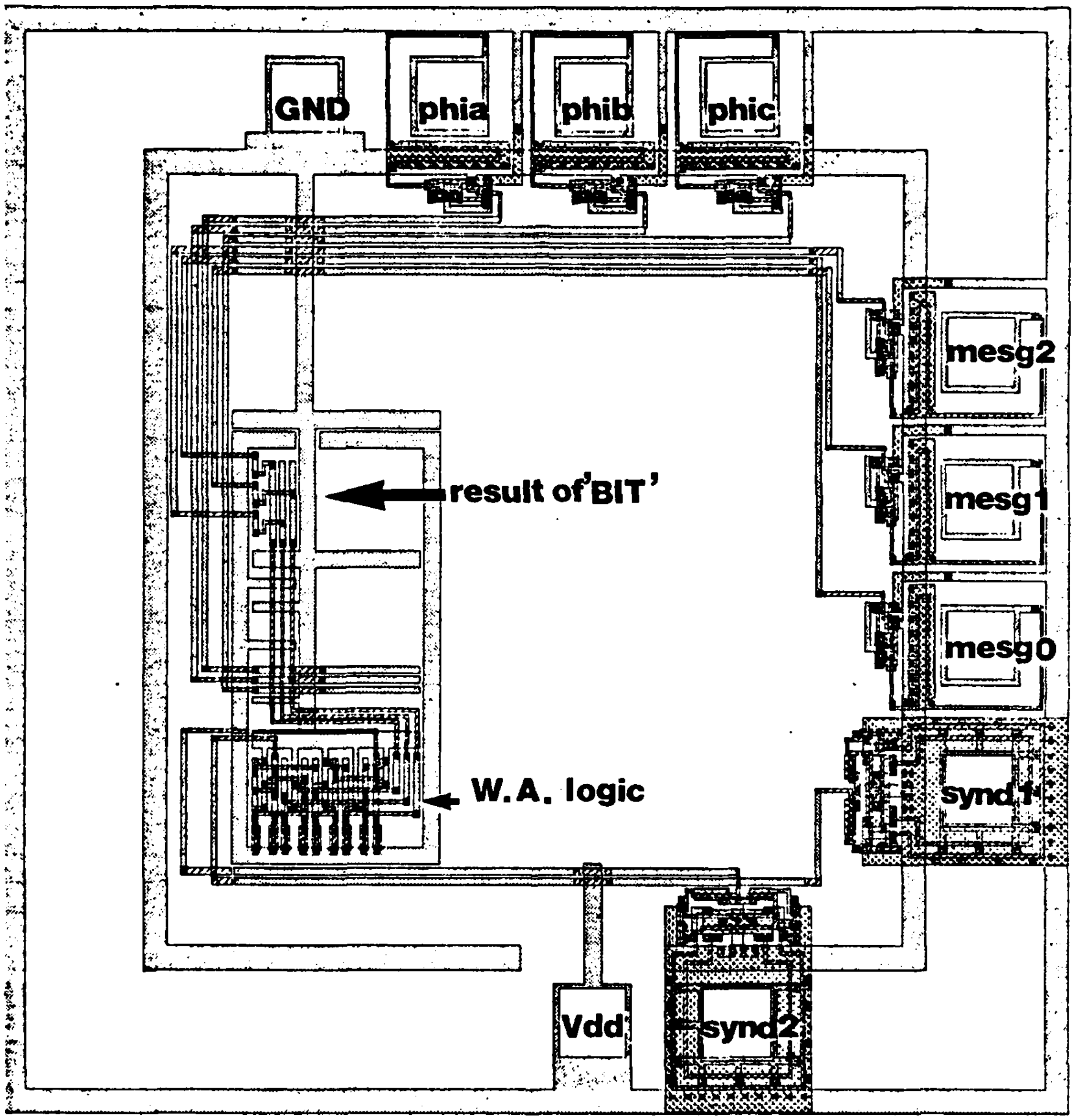


Figure 6.3 Ham3.cif

3. <u>Expanded Prototype</u>

The three bit Hamming error detector is the trivial case. The decision is in favor of the winning bits ("two out of three"), so the syndrome is not really necessary unless the check bits are wrong (a possibility for which the Hamming code allows).

The Hamming code is uniform in its protection, however; once encoded there is no difference between the message bits (i) and the check bits (c). This is important in checking longer words for errors. A seven bit message is checked as in the example given above. Elaborating on the prototype, Figure 6.4 shows the algorithm to generate the syndrome for a seven bit parallel error detector. This error detector requires a three bit syndrome to point at one of the possible seven error bits in the message. Section A. above illustrates the syndrome generation process, and how the syndrome word points at the erroneous message bit. The resulting cifplot is shown in Figure 6.5, and the structure is similar to the Weinberger array for the three-bit error detector.

It is good practice to expand the algorithm in steps, instead of going directly from the prototype to the final design. Unexpected results can be dealt with better if this approach is followed.

```
;HAM7.MAC
;A MacPitts algorithm to implement a 7 bit message error
;correction chip. The Hamming method is used. Four of the
;7 bits are data bits, 3 of the 7 are parity check positions.

(program ham7 1
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def msg port input (5 6 7 8 9 1Ø 11))
(def synd1 signal output 12)
(def synd2 signal output 13)
(def synd3 signal output 14)
(def 15 power)

;The Hamming method uses parity checks over bit positions
;1,3,5,and 7 to set the lsb of the syndrome,
;checks over positions 2,3,6,and 7 to set the middle synd bit,
;and checks over positions 4,5,6, and 7 to set the msb of the
;syndrome. The value of the syndrome indicates the bit error
;position in the 7 bit message.

(always

;set lsb of syndrome:
(cond
     ((parity (bit Ø msg) (bit 2 msg) (bit 4 msg) (bit 6 msg))
     (setq synd1 t ))
     (t
     (setq synd1  f)))

;set middle bit of syndrome:
(cond((parity (bit 1 msg) (bit 2 msg) (bit 5 msg) (bit 6 msg))
     (setq synd2 t ))
     (t
     (setq synd2 f )))

;set msb of syndrome:
(cond((parity (bit 3 msg) (bit 4 msg) (bit 5 msg) (bit 6 msg))
     (setq synd3 t ))
     (t
     (setq synd3 f )))       )      )
```

Figure 6.4 Ham7.mac

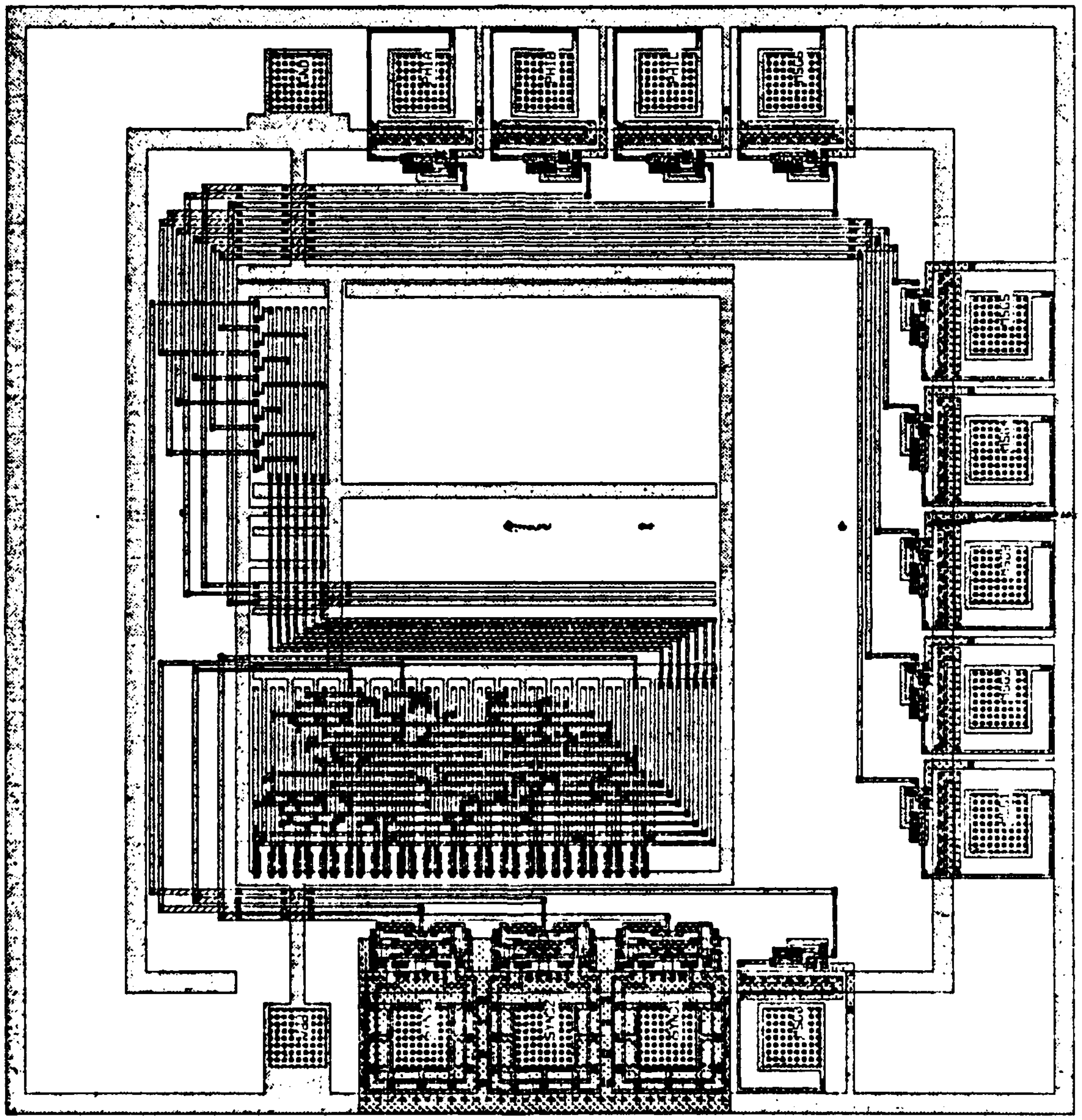


Figure 6.5 Ham7.cif

4. <u>Error Detector</u>

The desired algorithm is to uniformly detect errors in a 15 bit message. Remembering the surprising inability of MacPitts to compile a six input/one output gate in the data path, a test algorithm was written for the larger message. Figure 6.6 is the algorithm to detect errors in an 15 bit encoded message. The syndrome bits are determined from the parity checks as follows.

syndrome	message bit check positions
synd1	0 2 4 6 8 10 12 14
synd2	1 2 5 6 9 10 13 14
synd3	3 4 5 6 11 12 13 14
synd4	7 8 9 10 11 12 13 14

The single error detection scheme requires four bits to select the message bit for correction, thus the four bit syndrome. Synd1 is the lsb and synd4 is the msb of the Boolean syndrome word. Figure 6.7 shows the cifplot resulting from ham15.mac. The structure is predictably similar to ham7.cif and ham3.cif (Figure 6.3, Figure 6.5). This algorithm serves as the archetype (chief model, as opposed to prototype, first model) for the error detector. The error detector is half of the solution, the other half is correction of the errors. The detection is feasible, as proven by this algorithm.

Table 6.1 shows a comparison between the three error detectors.

```
;HAM15.MAC
;A MacPitts algorithm to implement an 11  bit message error
;correction chip. The Hamming method is used. 11  of the
;15 bits are data bits, 4 ~~of the~~ 11 are parity check positions.
(program ~~ham11 15~~
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def msg port input (5 6 7 8 9 10 11 12 13 14 15 16 17 18 19))
(def synd1 signal output 20)
(def synd2 signal output 21)
(def synd3 signal output 22)
(def synd4 signal output 23)
(def 24 power)
(always
;set lsb of syndrome:
(cond
     ((parity (bit 0 msg) (bit 2 msg) (bit 4 msg) (bit 6 msg)
              (bit 8 msg) (bit 10 msg) (bit 12 msg) (bit 14 msg))
     (setq synd1 t ))
     (t
     (setq synd1  f)))
;set next bit of stndrome:
(cond((parity (bit 1 msg) (bit 2 msg) (bit 5 msg) (bit 6 msg)
              (bit 9 msg) (bit 10 msg) (bit 13 msg) (bit 14 msg))
     (setq synd2 t ))
     (t
     (setq synd2 f )))
;set next bit of syndrome:
(cond((parity (bit 3 msg) (bit 4 msg) (bit 5 msg) (bit 6 msg)
              (bit 11 msg) (bit 12 msg) (bit 13 msg) (bit 14 msg))
     (setq synd3 t ))
     (t
     (setq synd3 f )))
;set msb of syndrome:
(cond((parity (bit 7 msg) (bit 8 msg) (bit 9 msg) (bit 10 msg)
              (bit 11 msg) (bit 12 msg) (bit 13 msg) (bit 14 msg))
     (setq synd4 t))
     (t
     (setq synd4 f )))      )       )
```

Figure 6.6 Ham15.mac

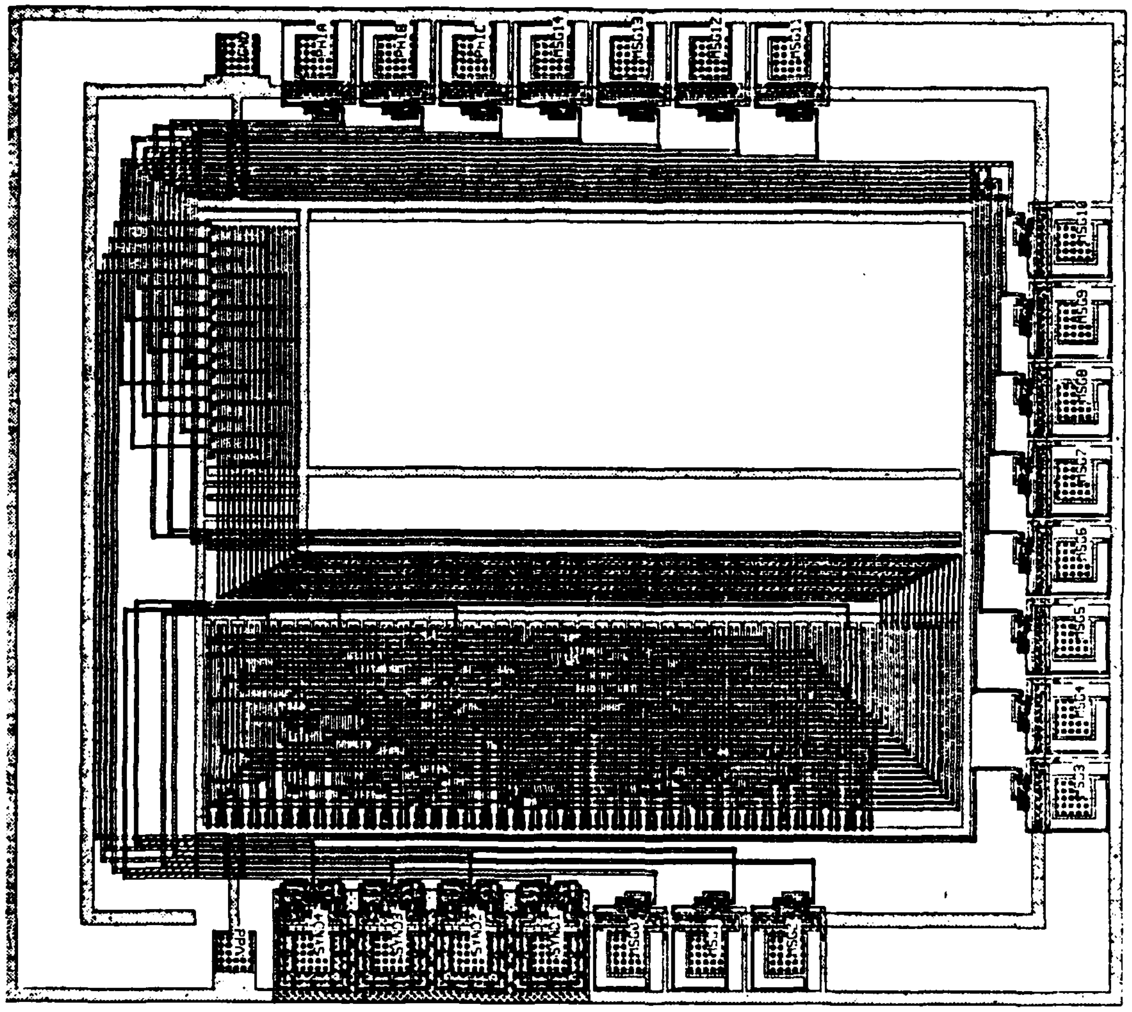


Figure 6.7 Ham15.cif

TABLE 6.1

THREE ERROR DETECTORS

	HAM3	HAM7	HAM15
Chip area [mm**2]	3.473	4.812	11.113
Control path area [mm**2]	2.75	1.918	8.025
Number pullups [in control]	9	31	71
Number pads	10	15	24
MacPitts pwr. [W]	.03194	.06094	.12265
Powest pwr. (avg) [W]	.02170	.03808	.06191
Powest pwr. (max) [W]	.04341	.07379	.11746
Max. delay [ns]	51.54	296.23	578.42
Max. frequency [MHz]	19.40	3.34	1.73
Cycles/result	1	—	—
Throughput [results/sec]	19.40M	3.34M	1.73M

So this method of parallel error detection appears feasible for word lengths less than 16 bits. The speed is fast due to the chosen single-state MacPitts architecture (ALWAYS = one PROCESS with zero stack depth, or for this purpose, a single-state FSM). These chips are unclocked circuits. The throughput is not a function of the clock rate, but depends on the signal propagation time from input to output. The propagation time sets the upper limit on throughput, and the capacitive leakage from the Weinberger array gates sets the lower limit on throughput. If the error detectors are used in a slow system, the outputs must therefore be latched to maintain valid logic levels. This is easily done with MacPitts, by SETQing the results to flags, and subsequently clocking the flags to output signal ports.

B. HAMMING METHOD 15/4 ERROR CORRECTOR

The previous section is only part of the story. Having located the error bit in the message, it must now be corrected. The decision of how to implement the error detector was a simple one, constrained by syntax. The error detector/corrector invites other methods of implementation.

1. Design Considerations

The message bit error is pointed at by the syndrome bits (the syndrome indicates the erroneous bit position). The error bit needs to be complemented, and the correct message results. The corrected message is then fed to the output ports. In this application, the extraneous check bits are discarded. The check bits (c) are used to encode the original message, and after reception and decoding they serve no purpose.

The message error detection and correction procedure can be reduced to three steps:

1. locate the error
2. complement the error bit
3. set the corrected output word bits

The first step is done with the error detection part of the algorithm. The second step is straightforward in MacPitts. Either the output bit is the input message bit (the correct message bit case), or else the output bit is the complement of the corresponding message bit (the incorrect message bit case). The checking is done with the COND form in MacPitts.

The third step involves discarding the check bits, setting the correct output bits to the corresponding input bit values, and sending the complement of the erroneous input bit to the corresponding output bit position.

2. Prototype Designs

Bit manipulations require Boolean data types, so flags and signals are used. The flags store the computed syndrome bits, and the signals are used for input and output. Figure 6.8 shows the MacPitts driver, ham3c.mac.

There are three COND statements in ham3c.mac. The first two determine the results of the message parity checks, as in the error detection algorithms. The last COND sets the single message bit according to the result of the parity checks. If fs1 (flag, synd1) is FALSE and fs0 is TRUE, then the message bit is incorrect. The output is then set to the complement of the input bit value. If the form under the last COND is FALSE, then either there is no error in the message, or the one of the two check bits is incorrect. In either case, the input message data bit is correct, so the output data bit (out0) is set to the lsb of the input message (msg0).

The format of the input is three symbols, two of which are check bits and one data (information) bit.

bit position	0	1	2
bit function	c	c	i

Only the last bit is returned from the error correction routine, the two check bits (inserted in the encoding of the message) are useless at this point. The last bit is the result of the error correction process, and is also the output of the prototype design. The algorithm (ham3c.mac) has the syndrome bits declared as output signals. This is considered good programming form (MacPitts being both a language and a silicon compiler), and allows troubleshooting the algorithm at run time. The syndrome outputs are unnecessary for the error corrector chip, and are deleted after verification of the algorithm in the command interpreter.

The resulting cifplot is Figure 6.9. The BIT organelles are absent, but two data path organelles corresponding to the flags fs1 and fs0 are instantiated. These are the storage elements for the computed syndrome

```
;HAM3C.MAC
;MacPitts algorithm for single-error detection & correction.
;This algorithm serves as a paradigm for the Hamming single
;error detection and correction problem.
(program haml 3
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
;msg(n) :  the input datum and 2 parity check bits
;outØ   :  the corrected datum
;synd(n):  the bit-checked Hamming error syndromes
;fs(n)  :  integer storage flags for the syndrome states
(def msg2 signal input 5)
(def msg1 signal input 6)
(def msgØ signal input 7)
(def outØ signal output 8)
(def synd1 signal output 9)
(def syndØ signal output 1Ø)
(def fsØ flag)
(def fs1 flag)
(def 11 power)
(always            ;a 1 state FSM

(cond              ;set the lsb of the error-bit syndrome:
     ((parity msgØ  msg2 )
     (setq syndØ t ) (setq fsØ t)  )
     (t
     (setq syndØ f ) (setq fsØ f)  ))
(cond              ;set the msb of the error-bit syndrome:
     ((parity msg1  msg2 )
     (setq synd1 t ) (setq fs1 t)  )
     (t
     (setq synd1 f ) (setq fs1 f)  ))
(cond              ;the fs(n) flag states determine whether
                   ;the output datum requires correction.
     ((and (not fs1) fsØ)
     (setq outØ (not msgØ)) )
     (t
     (setq outØ msgØ)))
     ))
```

Figure 6.8 Ham3c.mac

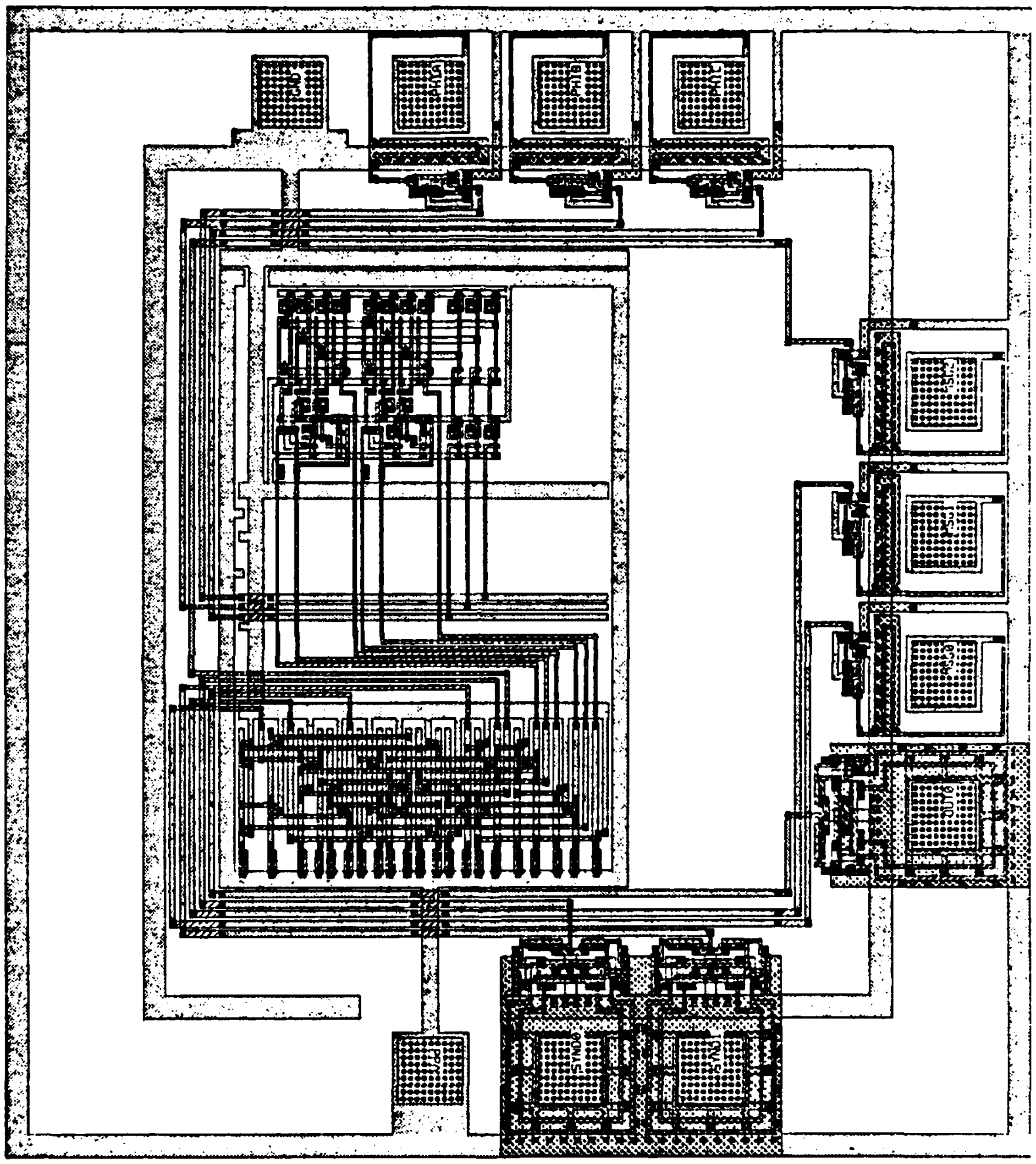


Figure 6.9 Ham3c.cif

values. The Weinberger array writes to and reads from these flags, as the algorithm suggests. An implication of this hybrid (data path and control path) structure is slower speed. This does not necessarily denote slower throughput, but slower signal speed across the logic circuitry.

To the right of the two flags is a bank of three dual cascaded vertical inverters. This structure performs a function analogous to what the clock drivers do for data path registers (superbuffering and sequencing of the three phases).

Just as the error detector was tested for the three bit, seven bit, and 15 bit cases, so is the error corrector tested next for the case of a seven bit message (the error corrector incorporates the error detector in its logic).

This section suggests a method whereby the designer can optimize the MacPitts chip. Three solutions to the error detection/correction problem are considered. Each is investigated, and the best solution is chosen as the archetype for the final 15 bit error corrector chip. The archetype is chosen on a seven bit basis instead of the simpler three bit chip. The seven bit error detector/correctors require more time to design and analyze, but their performance is more representative of the desired chip's than is the three bit detector/corrector.

The first method is an elaboration on ham3c.mac. The algorithm is shown in Figure 6.10, and the cifplot is

Figure 6.11. This algorithm uses three flags (fs0, fs1, and fs2) to store the individual syndrome bits. The syndrome bits are subsequently tested in the Weinberger array, and used to selectively set the four output bits of the corrected message (out6, out5, out4, and out2). This solution has the advantage of clarity, and the disadvantage of slowness due to the hybrid structure and poly run lengths. In comparing this algorithm to Figure 6.8 (ham3c.mac), it can be inferred that the number of COND statements in the error detection part of the algorithm is always the same as the number of parity checks needed. Similarly, the number of CONDs in the error correction part equals the number of output data bits.

This version of the chip requires two clock cycles to produce an output (write the error syndromes to the flags, then read the flags to determine the correct output). The throughput is 318,180 results/sec. A result is taken to be a corrected data word, in this case, a four-bit word.

Figure 6.12 shows an alternate solution, ham7cs.mac. This algorithm replaces the three flags with internal signals, is0, is1, and is2. Internal signals in MacPitts have the advantage of not requiring time-consuming storage operations. This architecture reduces the error corrector to a combinational logic structure, implemented in the control path due to syntax (all Boolean forms). The algorithm has a similar structure to the previous one which used flags to store the syndromes (Figure 6.10). There are three COND's to set the syndrome, and four CONDs to set the output word. The question of internal timing arises: will MacPitts have the syndrome ready in time for the output word setting? The answer is yes, because the algorithm executes sequentially in the order written in the absence of parallelizing forms (COND, PAR, PROCESS).

This algorithm is faster than the previous one also. The throughput is 2,034,000 words/sec, almost six times as fast as the chip using flags to store the syndrome.

Another solution considers the PAR form for paralleling the CONDS. An increase in speed results if the three CONDs which set the syndrome are paralled, and then the four CONDs which set the output are paralled with PAR. The throughput of this chip is 2,208,000 words/sec, slightly faster than the chip without PARs around the CONDs. This translates into larger structure (Table 6.2). Figure 6.14 is the MacPitts driver, ham7cr.mac, and Figure 6.15 is the cifplot.

This version of the error detector/corrector is the archetype (chief example) for the 15 bit error detector/corrector. It was developed based on the three bit prototype (Figure 6.8), refined , tested with the MacPitts interpreter and Crystal, and is considered the optimal MacPitts parallel-architecture solution for the seven bit correction problem. It serves as the model for building the

```
;HAM7Cfth.MAC
;Hamming 7 bit message error corrector, FLAGS for syndromes
(program ham7cfth  1
(def 1 ground)(def 2 phia)(def 3 phib)(def 4 phic)
(def msgØ signal input 5)(def msg1 signal input 6)
(def msg2 signal input 7)(def msg3 signal input 8)
(def msg4 signal input 9)(def msg5 signal input 1Ø)
(def msg6 signal input 11)
(def out6  signal output 12)(def out5 signal output 13)
(def out4  signal output 14)(def out2 signal output 15)
(def fs2 flag)     ;FLAGS store syndromes' states:
(def fs1 flag)
(def fsØ flag)
(def 16 power)
(always
;set lsb of syndrome:
(cond
     ((parity  msgØ  msg2  msg4  msg6)
     (setq fsØ t   )   )
     (t
     (setq fsØ f   )   ))
;set middle bit of syndrome:
(cond((parity msg1  msg2 msg5  msg6)
     (setq fs1 t   )   )
     (t
     (setq fs1 f   )   ))
;set msb of syndrome:
(cond((parity  msg3  msg4  msg5  msg6)
     (setq fs2 t   )   )
     (t
     (setq fs2 f   )   ))
;The erroneous MESSAGE bits are corrected
;Check data bit 2 (msg bit 3):
(cond
   ((and (not fs2) fs1  fsØ    )
   (setq out2 (not msg2) )     )
   (t
   (setq out2 msg2))           )
;Check data bit 4 (msg bit 5):
(cond
   ((and fs2 (not fs1)  fsØ    )
   (setq out4 (not msg4) )     )
   (t
   (setq out4 msg4))           )
;Check data bit 5 (msg bit 6):
(cond
   ((and fs2 fs1 (not fsØ)     )
   (setq out5 (not msg5) )     )
   (t
   (setq out5 msg5))           )
;Check data bit 6 (msg bit 7):
(cond
   ((and fs2 fs1  fsØ          )
   (setq out6 (not msg6) )     )
   (t
   (setq out6 msg6))           )))
```

Figure 6.10 Ham7cf.mac

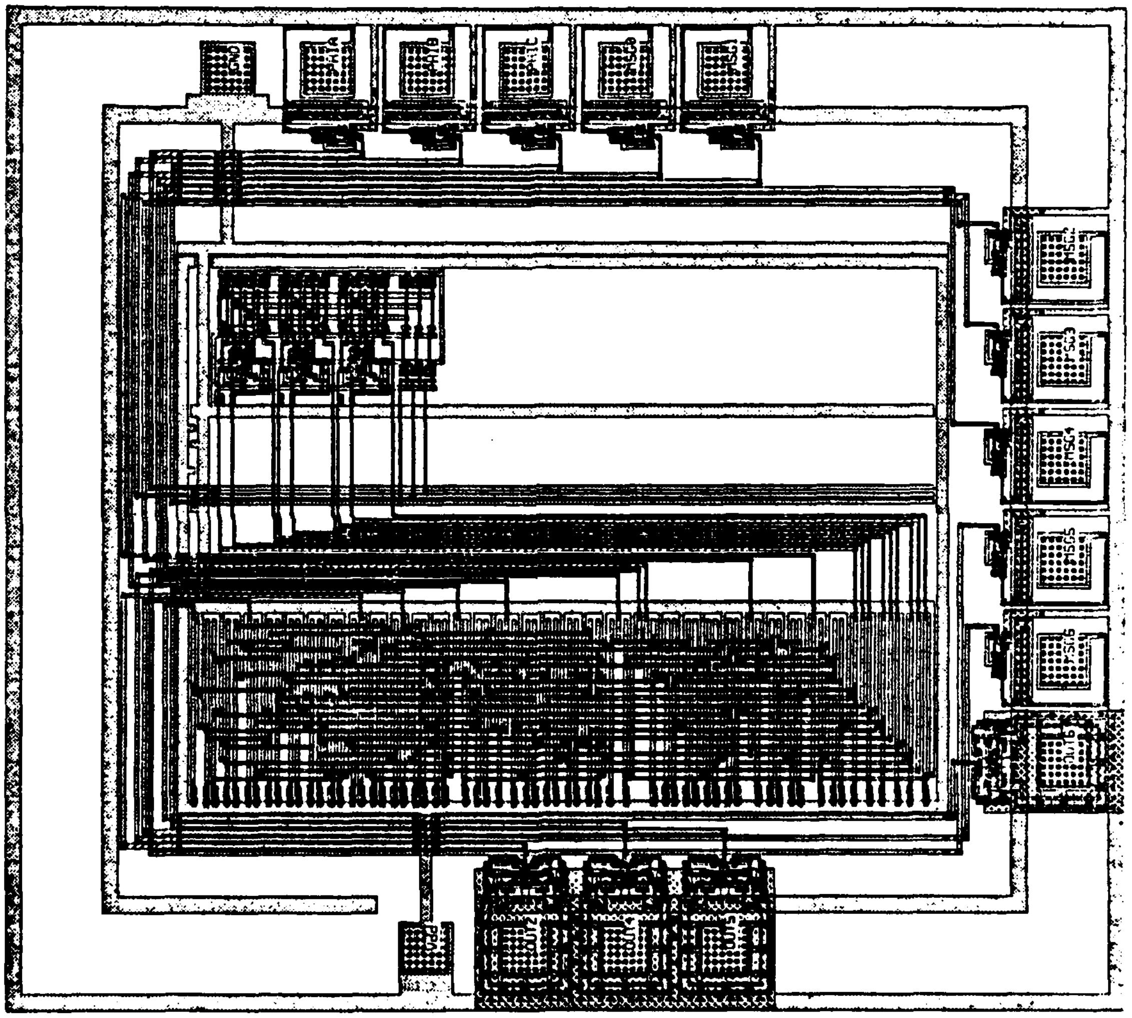


Figure 6.10 Ham7cf.cif

```
;HAM7Cs.MAC
;Hamming 7 bit message error corrector,SIGNALS
(program ham7cs  1
(def 1 ground)(def 2 phia)(def 3 phib)(def 4 phic)
(def msgØ signal input 5)(def msg1 signal input 6)
(def msg2 signal input 7)(def msg3 signal input 8)
(def msg4 signal input 9)(def msg5 signal input 1Ø)
(def msg6 signal input 11)
(def out6  signal output 12)(def out5 signal output 13)
(def out4  signal output 14)(def msg2 signal output 15)
;3 signals needed to pass the syndrome's bits:
(def is2 signal internal)  ;use SIGNALS instead of FLAGS:
(def is1 signal internal)
(def isØ signal internal)
(def 17 power)
(always
;set lsb of syndrome:
(cond
     ((parity  msgØ  msg2  msg4  msg6)
     (setq isØ t    )    )
     (t
     (setq isØ f    )    ))
;set middle bit of syndrome:
(cond((parity msg1  msg2 msg5  msg6)
     (setq is1 t    )    )
     (t
     (setq is1 f    )    ))
;set msb of syndrome:
(cond((parity  msg3  msg4  msg5  msg6)
     (setq is2 t    )    )
     (t
     (setq is2 f    )    ))
;Check data bit 2 (msg bit 3):
(cond
   ((and (not is2) is1  isØ      )
   (setq out2 (not msg2) )       )
   (t
   (setq out2 msg2))             )
;Check data bit 4 (msg bit 5):
(cond
   ((and is2 (not is1)  isØ      )
   (setq out4 (not msg4) )       )
   (t
   (setq out4 msg4))             )
;Check data bit 5 (msg bit 6):
(cond
   ((and is2 is1 (not isØ)       )
   (setq out5 (not msg5) )       )
   (t
   (setq out5 msg5))             )
;Check data bit 6 (msg bit 7):
(cond
   ((and is2 is1  isØ            )
   (setq out6 (not msg6) )       )
   (t
   (setq out6 msg6))             )))
```

Figure 6.12 Ham7cs.mac

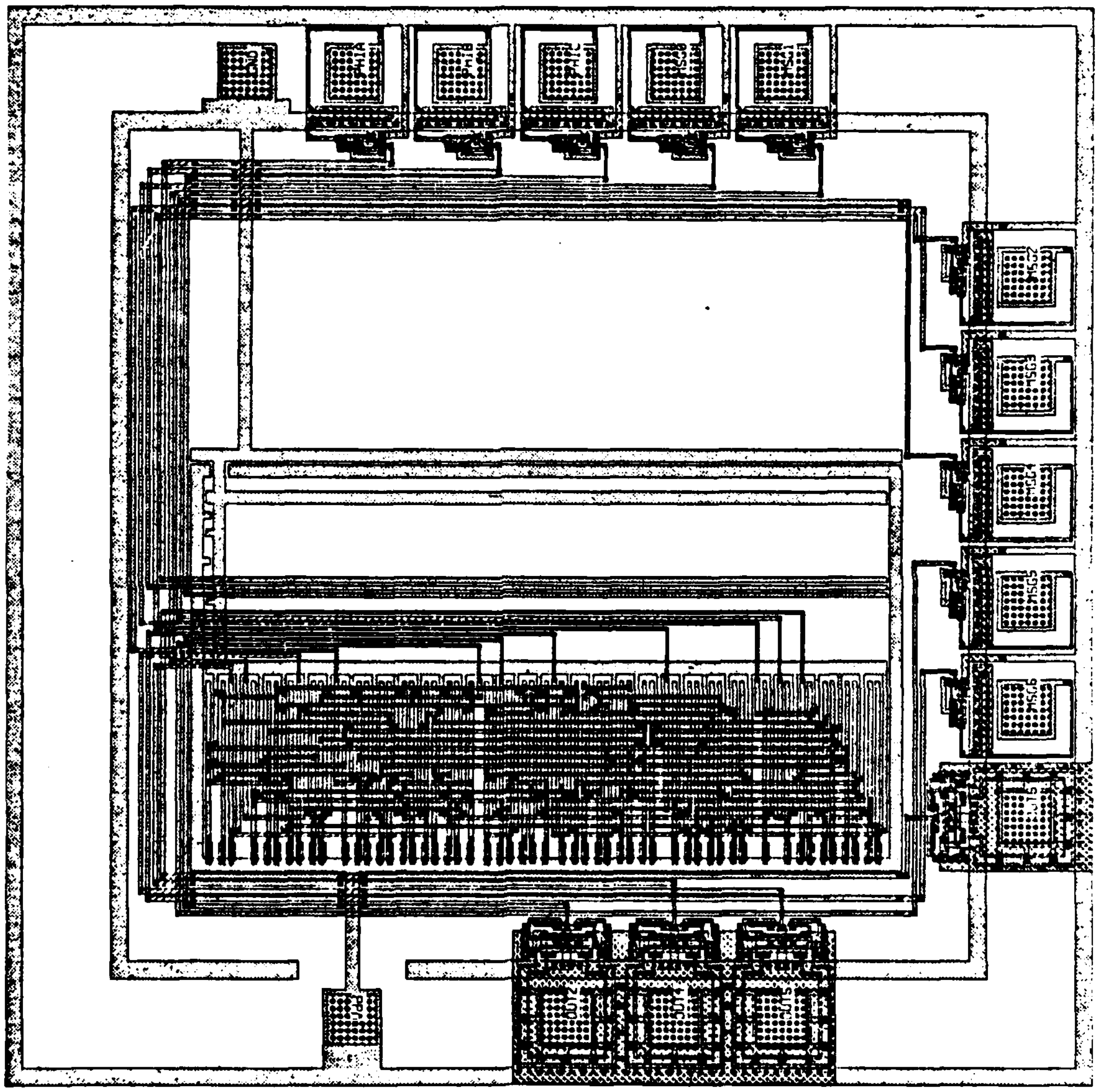


Figure 6.13 Ham7cs.mac

```
;HAM7Cr.MAC
;Hamming 7 bit message error corrector, using PAR
(program ham7cr  1
(def 1 ground)(def 2 phia)(def 3 phib)(def 4 phic)
(def msgØ signal input 5)(def msg1 signal input 6)
(def msg2 signal input 7)(def msg3 signal input 8)
(def msg4 signal input 9)(def msg5 signal input 1Ø)
(def msg6 signal input 11)
(def out6  signal output 12)(def out5 signal output 13)
(def out4  signal output 14)(def out2 signal output 15)
;3 signals needed to pass the syndrome's bits:
(def is2 signal internal)
(def is1 signal internal)
(def isØ signal internal)
(def 17 power)
(always                 ;do every clk cycle
;set lsb of syndrome:
(par
(cond                   ;PARallel parity checking, setting
     ((parity  msgØ  msg2  msg4  msg6)
     (setq isØ t   )   )
     (t
     (setq isØ f   )   ))
;set middle bit of syndrome:
(cond((parity msg1  msg2 msg5  msg6)
     (setq is1 t   )   )
     (t
     (setq is1 f   )   ))
;set msb of syndrome:
(cond((parity  msg3  msg4  msg5  msg6)
     (setq is2 t   )   )
     (t
     (setq is2 f   )   ))   )
;Check data bit 2 (msg bit 3):
(par
(cond
   ((and (not is2) is1  isØ    )
   (setq out2 (not msg2) )     )
   (t
   (setq out2 msg2))           )
;Check data bit 4 (msg bit 5):
(cond
   ((and is2 (not is1)  isØ    )
   (setq out4 (not msg4) )     )
   (t
   (setq out4 msg4))           )
;Check data bit 5 (msg bit 6):
(cond
   ((and is2 is1 (not isØ)     )
   (setq out5 (not msg5) )     )
   (t
   (setq out5 msg5))           )
;Check data bit 6 (msg bit 7):
(cond
   ((and is2 is1  isØ          )
   (setq out6 (not msg6) )     )
   (t
   (setq out6 msg6))           ))))
```

Figure 6.14 Ham7cr.mac

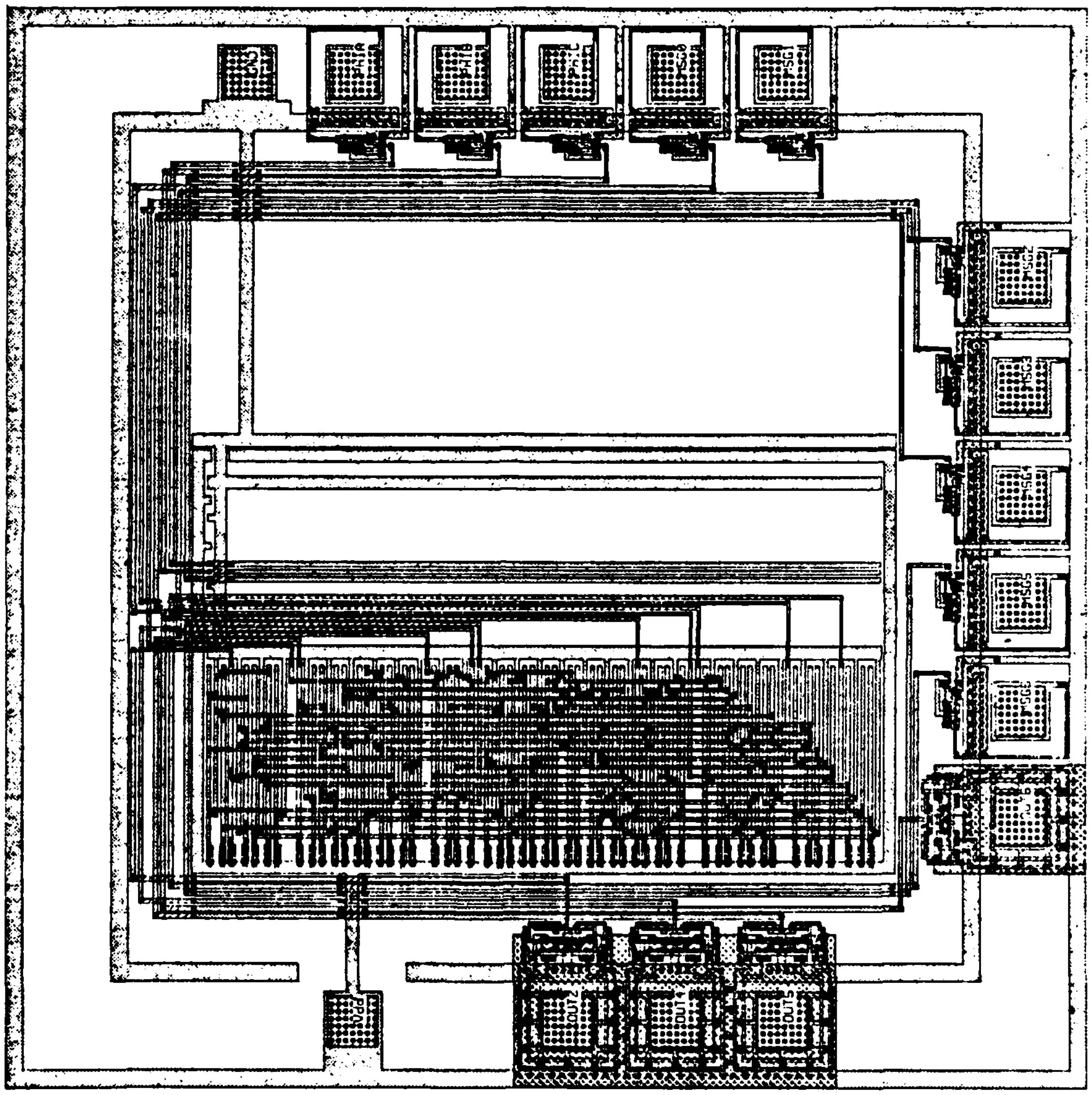


Figure 6.15 Ham7cr.cif

15 bit machine (the seven bit model is easier to analyze in the interpreter, and with Crystal and Esim).

It is impractical to do the preceeding design process beginning with a 15 bit machine. The 15 bit message cannot be tested in the interpreter (all the inputs and outputs will not fit on the VT-100 screen), and Caesar and Crystal analysis is far more complicated with large structures. It is better to optimize with a smaller model, and then extend the results to achieve the desired chip.

Table 6.2 is a parametric comparison of the three Hamming error detector/corrector chips. The reason for the choice of ham7cr.mac is clear from previous discussion and these statistics.

TABLE 6.2

CHIP PARAMETRIC COMPARISON

	HAM7Cf	HAM7Cs	HAM7Cr
Area [mm**2]	7.003	6.305	6.187
Power [W]	.102	.0931	.0931
Delay [ns]	1581.37	491.64	452.94
Speed [MHz]	.6324	2.034	2.208
Cycles/res.	2	1	1
Throughput [res/s]	.316M	2.2034M	2.208M
Speed/area [MHz/mm**2]	.0903M	.3226M	.3579M
Density [tran/mm**2]	53.6	45.7	46.6

The reason for the choice of ham7cr as the model is seen in Table 6.2. The chip (Ham7cr) is smaller and faster than its predecessors. It has the highest throughput of all the seven bit correctors. The result of using the PAR form is seen by comparing the speed/area ratios of ham7cs and ham7cr. PAR translates into more decisions done simultaneously, and the decisions are done faster (speed/area is greater). The result of storing the syndrome bits in flags (ham7cf) is shown in its comparatively low throughput and low speed/area figures.

A functional summary of the three prototype candidate algorithms (flowcharts and resulting floorplans) is given in Figures 6.16 - 6.21.

4. <u>Hamming 15/4 Error Corrector</u>

The 15 bit error corrector is designed after the PARalled COND version of the ham7 algorithm, ham7cr.mac (Figure 6.14). As explained above, the number of CONDs expected is the sum of the number of syndrome bits and the number of corrected data bits out. There are four syndrome bits for the 15/4 code, and 11 corrected data bits out, for a total of 15 CONDs in the algorithm. Figure 6.22 shows ham15dc.mac. The algorithm structure is similar to ham7, except for the pin naming which has been shortened to make it easier to enter the data for analysis (Crystal, Caesar labels, esim). There are four parity checks across the bits as described in the paragraph on error detection. The parity

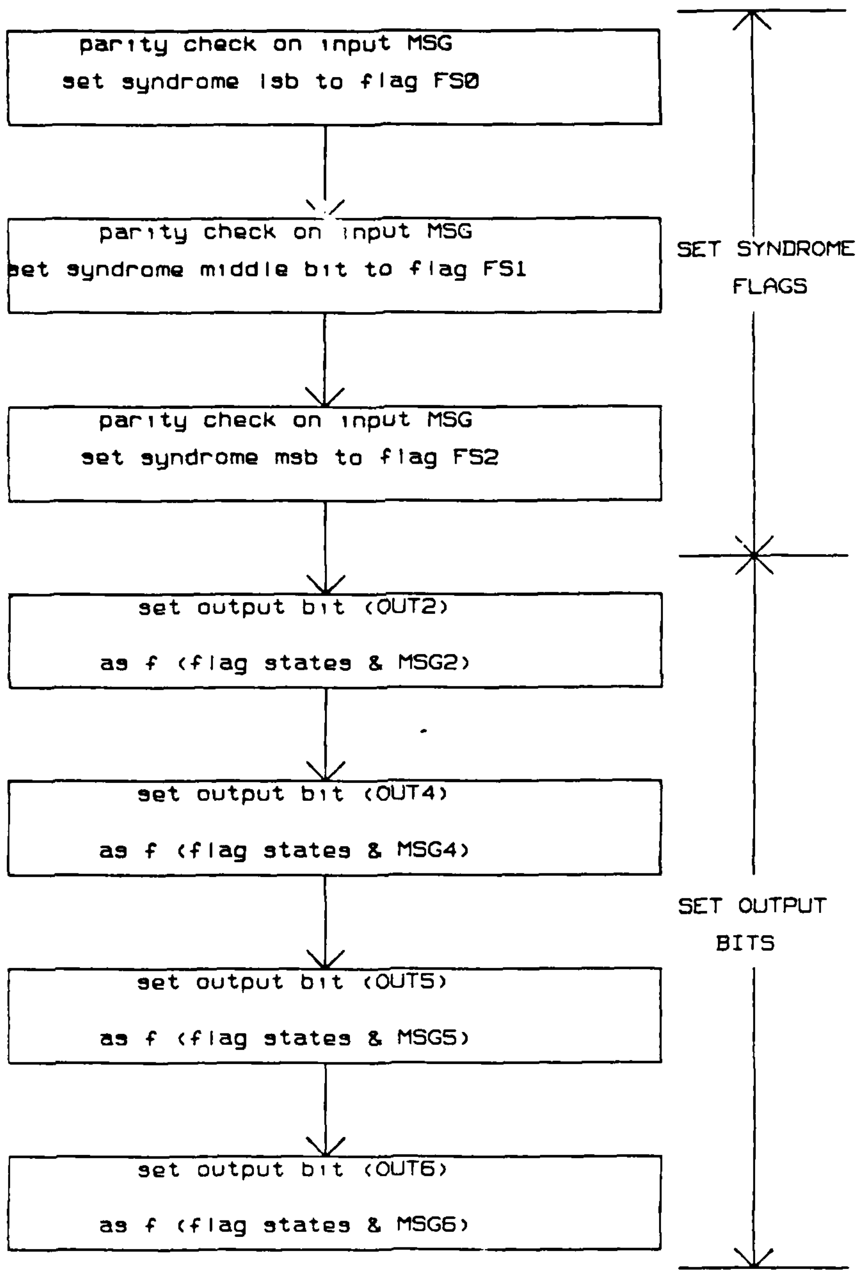

parity check on input MSG
set syndrome lsb to flag FS0
parity check on input MSG
set syndrome middle bit to flag FS1
parity check on input MSG
set syndrome msb to flag FS2
set output bit (OUT2)
as f (flag states & MSG2)
set output bit (OUT4)
as f (flag states & MSG4)
set output bit (OUT5)
as f (flag states & MSG5)
set output bit (OUT6)
as f (flag states & MSG6)
SET SYNDROME FLAGS
SET OUTPUT BITS

Figure 6.16 Ham7cf Flowchart

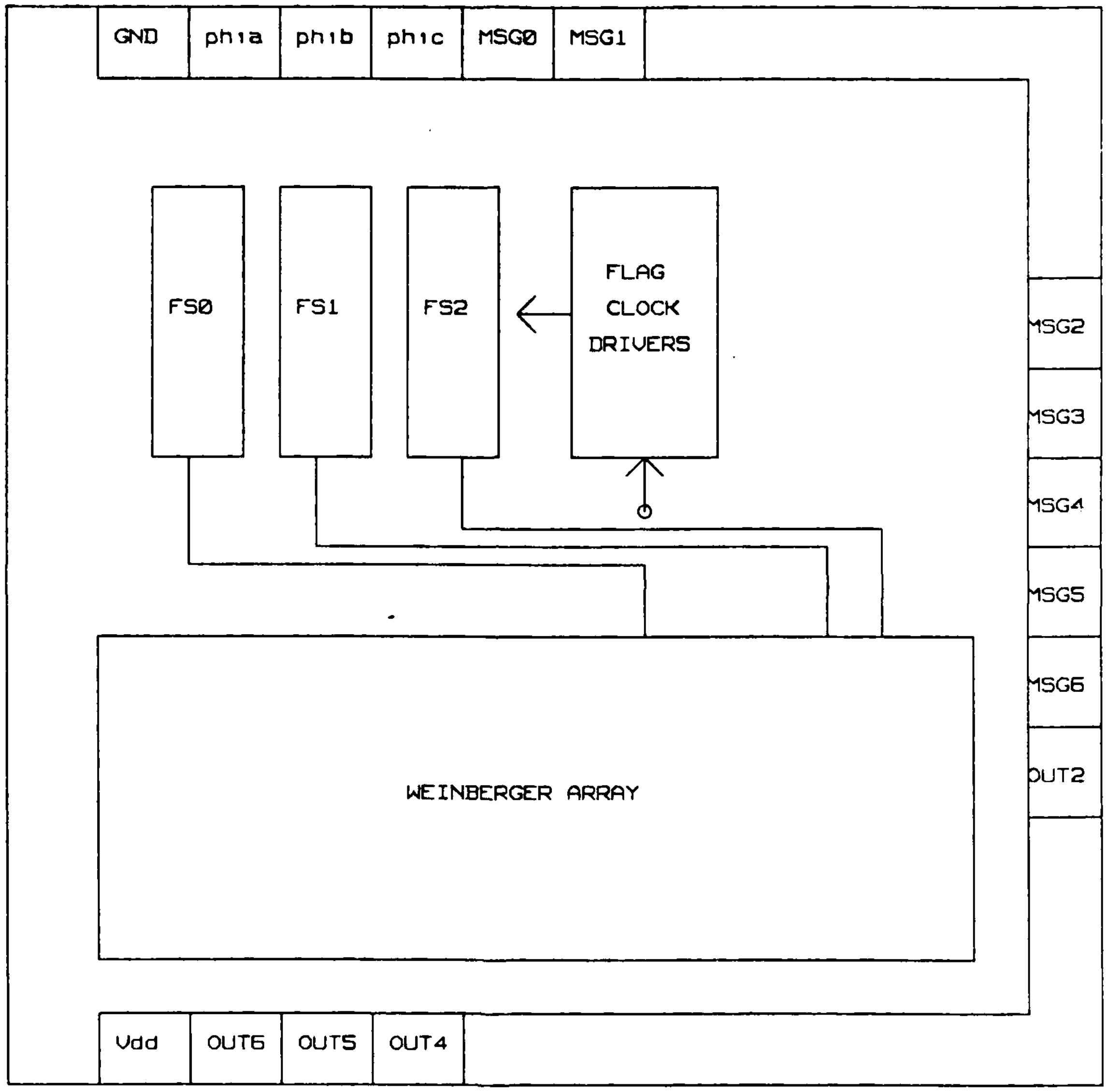


Figure 6.17 Ham7cf Floorplan

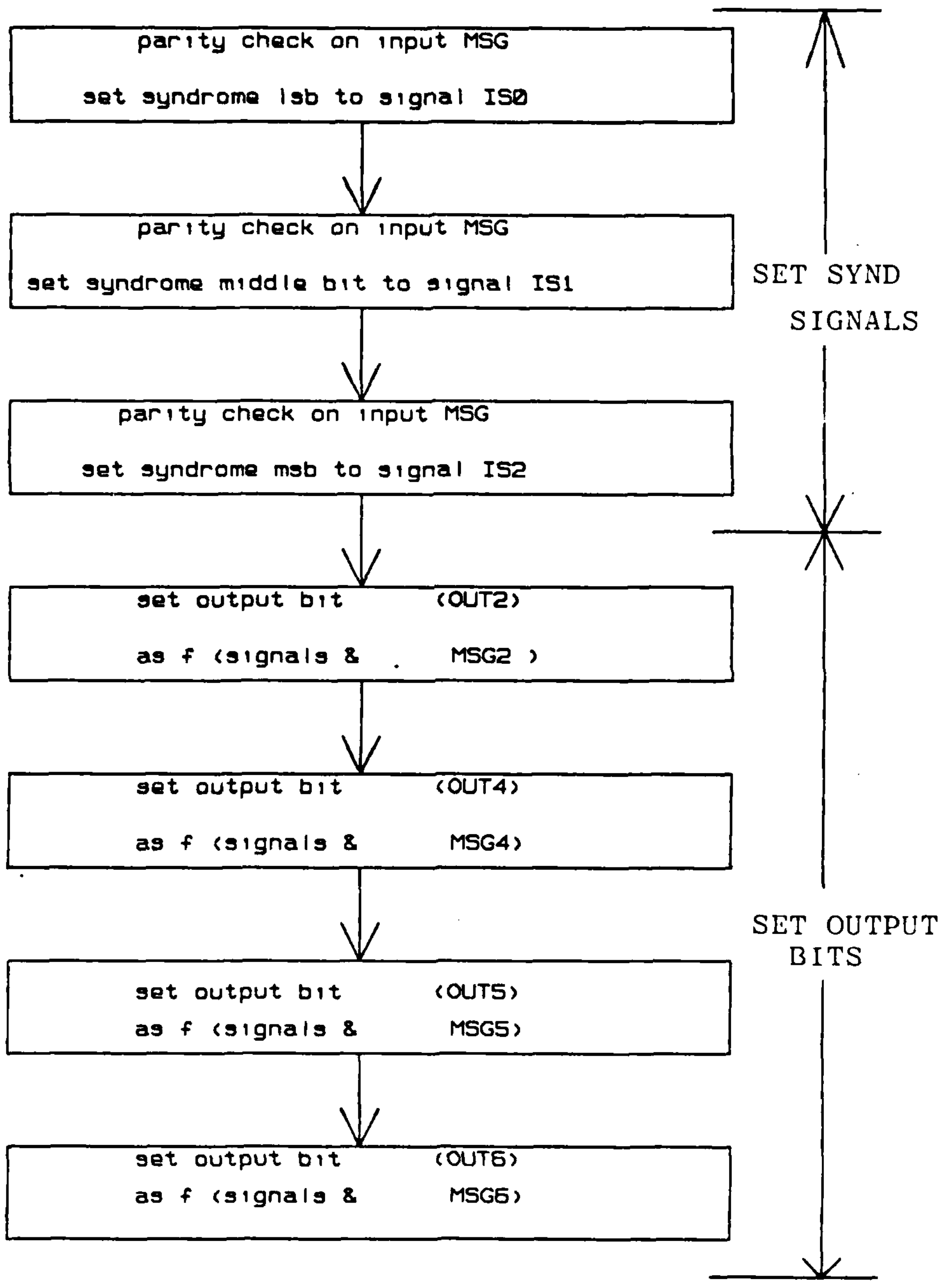


Figure 6.18 Ham7cs Flowchart

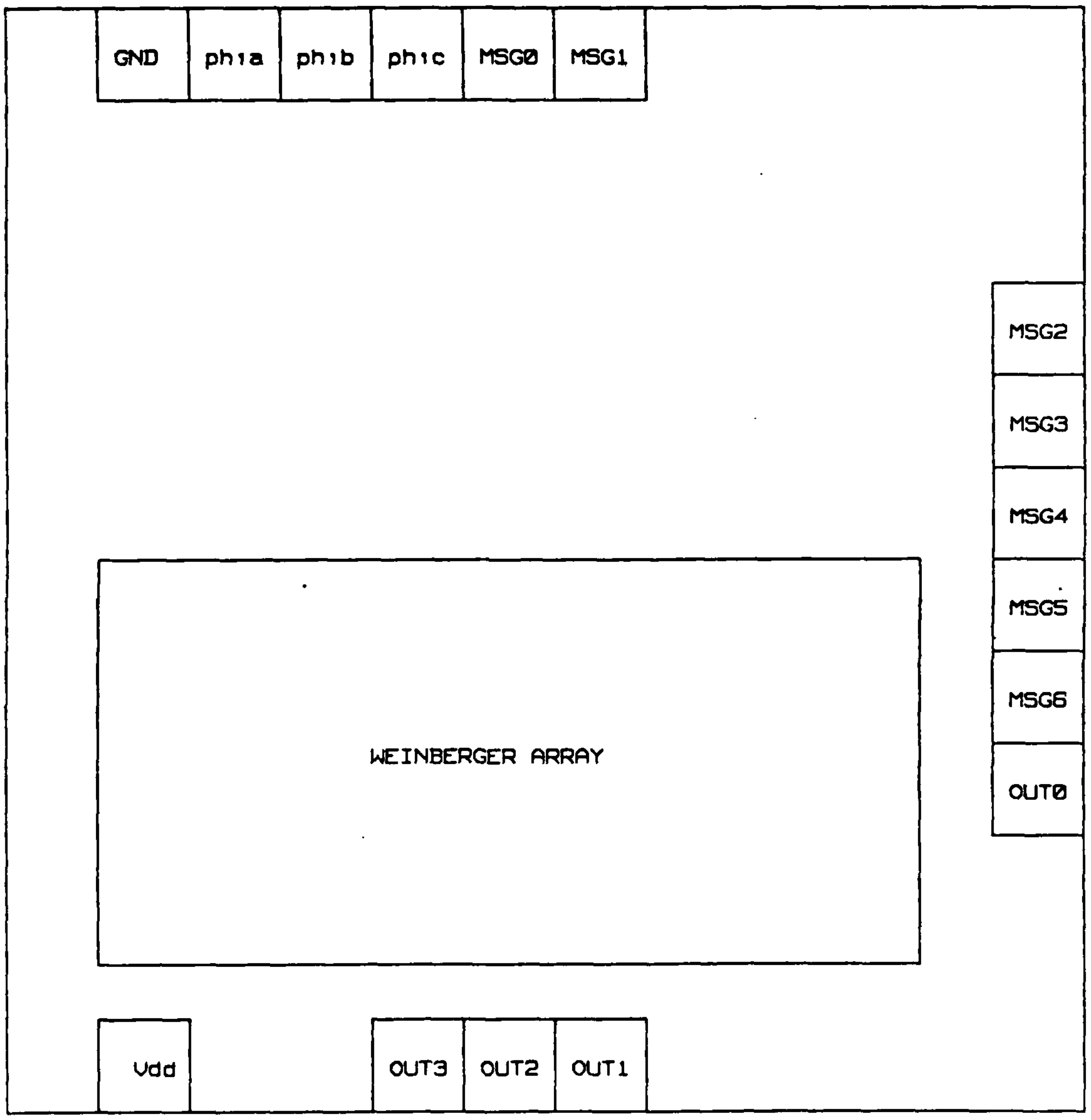


Figure 6.19 Ham7cs Floorplan

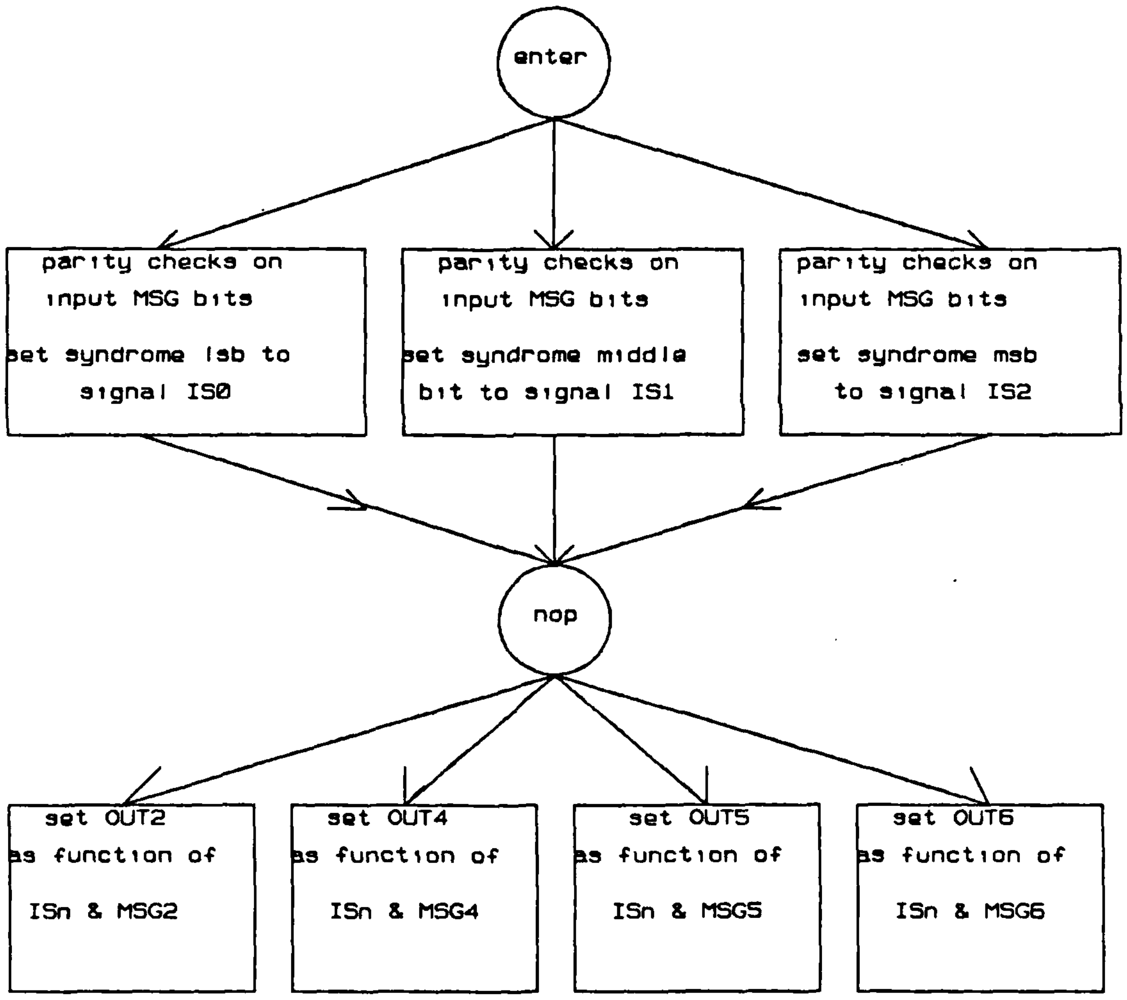

enter
parity checks on input MSG bits
set syndrome lsb to signal IS0
parity checks on input MSG bits
set syndrome middle bit to signal IS1
parity checks on input MSG bits
set syndrome msb to signal IS2
nop
set OUT2 as function of ISn & MSG2
set OUT4 as function of ISn & MSG4
set OUT5 as function of ISn & MSG5
set OUT6 as function of ISn & MSG6

Figure 6.20 Ham7cr Flowchart

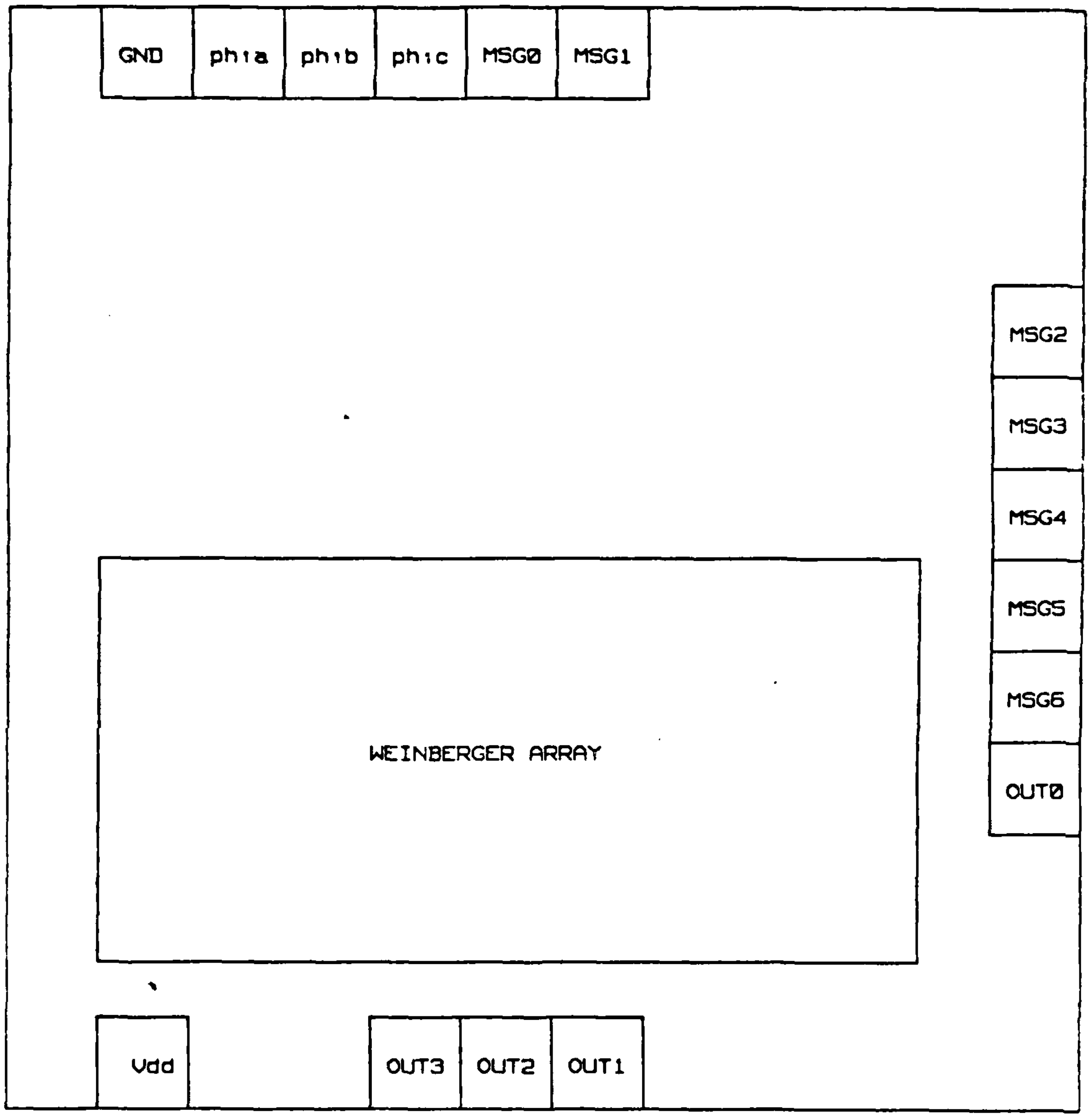


Figure 6.21 Ham7cr Floorplan

checks result in four syndrome internal signals. The internal signals translate to feedback within the Weinberger array. After the bit error is identified by the syndrome pattern, it is corrected. There are 11 CONDs which accomplish the bit-wise correction of the output word, one for each bit which is not an encoding bit (positions 0, 1, 3, and 7).

The algorithm compiled to cif, as expected. The size of the Weinberger array (155 columns) required a long time for compilation, approximately 3.5 hours (at night) on the VAX 11/780 at Naval Postgraduate School. The resulting labelled cifplot is shown in Figure 6.23. The circuit is an expansion of the seven bit Hamming error correctors, but larger. The seven bit chip has seven CONDs, the 15 bit chip has 15. The result of COND in the algorithm is NOR gates in the Weinberger array. The chip measures 5.1371 mm by 4.005 mm, for an area of 20.57 sq. mm. There are 238 pullup transistors, so the Powest-calculated power dissipation of 0.1229 W (average) is no surprise (MacPitts estimates the power consumption as 0.16086 W). The Powest estimated maximum dc power is 0.2321 W. Crystal timing analysis predicts a maximum delay of 1222.94 ns, for a maximum data rate of 818 kHz and therefore a maximum throughput of 818,000 results/sec (8,998,000 bits/sec). The circuit density is sparse, as seen in the cifplot, and the average density is approximately 37 transistors/sq. mm. The sparsity

```
;HAM15dc.MAC
;Hamming 15/4 error detector/corrector
(program ham15dc  1
(def 1 ground)(def 2 phia)(def 3 phib)(def 4 phic)
(def mØ  signal input  5 )(def m1 signal input 6  )
(def m2  signal input  7 )(def m3 signal input 8  )
(def m4  signal input  9 )(def m5 signal input 1Ø )
(def m6  signal input  11)(def m7 signal input 12 )
(def m8  signal input  13)(def m9 signal input 14 )
(def m1Ø signal input  15)(def m11 signal input 16)
(def m12 signal input  17)(def m13 signal input 18)
(def m14 signal input  19)
(def s14 signal output 2Ø)(def s13 signal output 21)
(def s12 signal output 22)(def s11 signal output 23)
(def s1Ø signal output 24)(def s9  signal output 25)
(def s8  signal output 26)(def s6  signal output 27)
(def s5  signal output 28)(def s4  signal output 29)
(def s2  signal output 3Ø)
(def 31 power)
(def isØ signal internal)(def is1 signal internal)
(def is2 signal internal)(def is3 signal internal)
(always
(par        ;PARallel syndrome setting:
;set lsb of syndrome:
(cond
     ((parity mØ m2 m4 m6 m8 m1Ø m12 m14)(setq isØ t))
     (t                                  (setq isØ f)))
;set middle bit of syndrome:
(cond
     ((parity m1 m2 m5 m6 m9 m1Ø m13 m14)(setq is1 t))
     (t                                  (setq is1 f)))
;set next bit syndrome:
(cond
     ((parity m3 m4 m5 m6 m11 m12 m13 m14)(setq is2 t))
     (t                                   (setq is2 f)))
;set msb syndrome:
(cond
     ((parity m7 m8 m9 m1Ø m11 m12 m13 m14)(setq is3 t))
     (t                                     (setq is3 f))))
;check & set output data bits:
(par             ;PARallel check/set operations:
;data bit 2 (m3)
(cond
     ((and (not is3) (not is2) is1 isØ)
     (setq s2 (not m2)))
     (t(setq s2 m2)))
;data bit 4 (m5)
(cond
     ((and (not is3) is2 (not is1) isØ)
     (setq s4 (not m4)))
     (t(setq s4 m4)))
;data bit 5 (m6)
(cond
     ((and (not is3) is2 is1 (not isØ))
     (setq s5 (not m4)))
     (t(setq s5 m5)))
;data bit 6 (m7)
(cond
```

Figure 6.22 Ham15dc.mac

```
          ((and (not is3) is2 is1 isØ)
          (setq s6 (not m6)))
          (t(setq s6 m6)))
     ;data bit 8 (m9)
     (cond
          ((and is3 (not is2) (not is1) isØ)
          (setq s8 (not m8)))
          (t(setq s8 m8)))
     ;data bit 9 (m1Ø)
     (cond
          ((and is3 (not is2) is1 (not isØ))
          (setq s9 (not m9)))
          (t(setq s9 m9)))
     ;data bit 1Ø (m11)
     (cond
          ((and is3 (not is2) is1 isØ)
          (setq s1Ø (not m1Ø)))
          (t(setq s1Ø m1Ø)))
     ;data bit 11 (m12)
     (cond
         ((and is3 is2 (not is1) (not isØ))
         (setq s11 (not m11)))
         (t(setq s11 m11)))
     ;data bit 12 (m13)
     (cond
         ((and is3 is2 (not is1) isØ)
         (setq s12 (not m12)))
         (t(setq s12 m12)))
     ;data bit 13 (m14)
     (cond
         ((and is3 is2 is1 (not isØ))
         (setq s13 (not m13)))
         (t(setq s13 m13)))
     ;data bit 14 (m15)
     (cond
         ((and is3 is2 is1 isØ)
         (setq s14 (not m14)))
         (t(setq s14 m14))))))
```

Figure 6.22 Ham15dc.mac (continued)

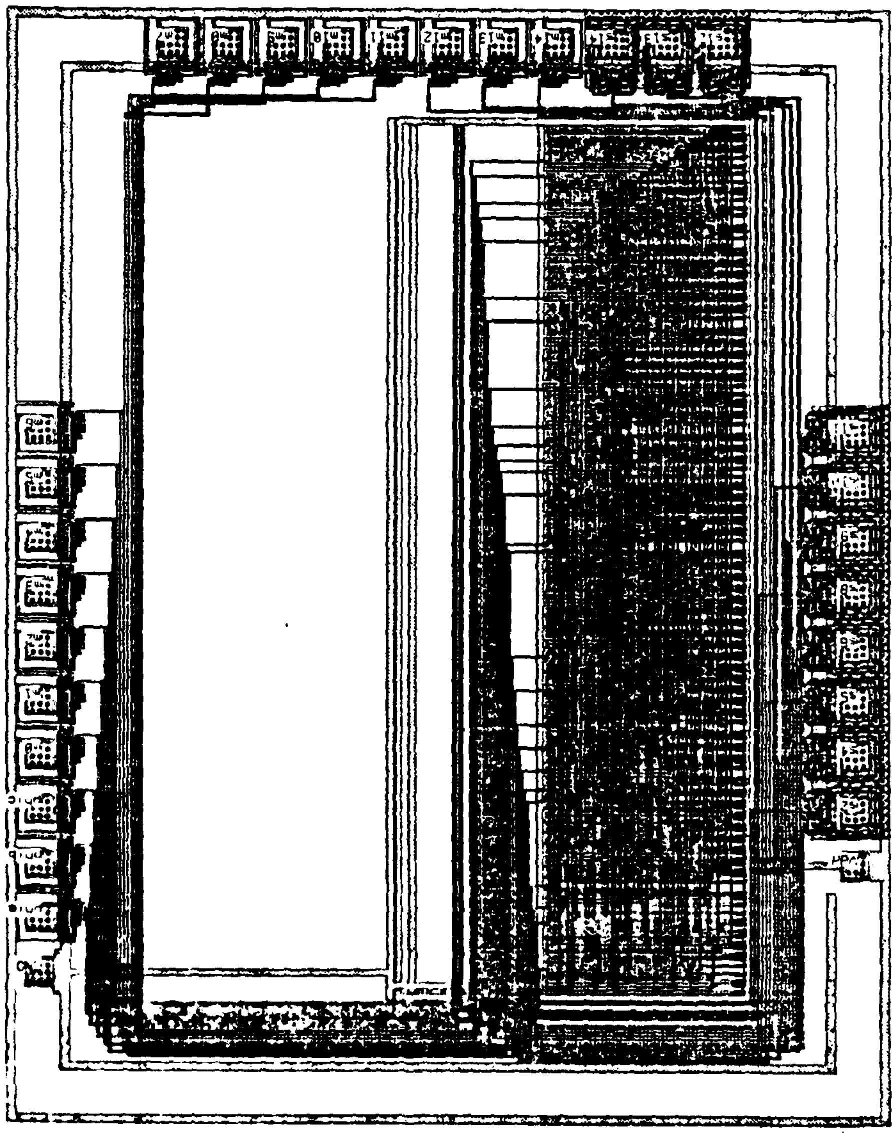


Figure 6.23 Ham15dc.cif

is due in part to the absence of a data path. If just the Weinberger array is considered, however, the circuit density is approximately 100 transistors/sq. mm. Appendix D contains the script recording of the compilation of ham15dc.mac.

The transistor densities given in Table 6.2 are derived from MacPitts chips. A comparison with standard library cells densities derived from Newkirk and Matthews [Ref. 12] may be illuminating.

TABLE 6.3

TRANSISTOR DENSITY COMPARISON

CIRCUIT	DENSITY [tran./mm**2]
Ham7Cf	54
Ham7Cs	46
Ham7Cr	47
CountUDRestore [Ref. 11:p. 79]	457
COUNT [Ref. 11:p. 67]	753
ALU [Ref. 11:p. 20]	616
ADDER [Ref. 11:p. 10]	691

So the MacPitts chips are far less dense than even the library macro cells. The Newkirk-Mathews cells only consider the cell itself, and not the chip, which was the basis on which the MacPitts densities were calculated.

Nevertheless, a density factor of 10 is a considerable difference (the MacPitts chips in this chapter are approximately 50% circuitry, and 50% white space, so a density factor of five is still significant).

VII. CONCLUSION

A. SUMMARY

This thesis has considered the effects of syntax on circuit structure in the MacPitts silicon compiler. The combinational logic structure is explicitly specified by syntax in the data path, and the appropriate behavior results. The circuit behavior is explicitly specified in the control path, and the combinational logic structure (a Weinberger array) results.

Combinational logic structures in the data path comprise adjoined MacPitts macros (organelles). Combinational logic structure in the control path, however, is always done in a Weinberger array. The poly runs internal and external to the Weinberger array make combinational logic operate slower there than in the equivalent circuit in the data path. Parallelism of logical functions is possible in MacPitts by using the COND and PAR forms. These paralleling forms usually equate to a speed/area tradeoff on the chip.

Sequential logic in MacPitts is implemented as a Mealy-type FSM. The state registers store the present state, and receive present input information from both the control path and the sequencer tail organelle. The data path width, as declared in the PROGRAM statement, determines the number of states possible for the FSM. This must be determined by the designer a priori, and explicitly stated in the PROGRAM statement. The long poly runs between the data path and control path cause a slow speed in the MacPitts FSM, as compared to the handcrafted equivalent. The 8:1 ratioed superbuffered input pads add to this slowness, because of the number of NOR gates one pad may have to drive in the Weinberger array.

The FSM architecture and its attendant Mealy sequencer organelles are implicitly specified by the PROCESS statement. Each process is an independent entity in MacPitts, with its own organelles and wires. Processes do not communicate internally with each other. The PROCESS form is another method of parallelism possible in MacPitts. All PROCESSES embraced by PROGRAM execute in parallel, at the speed of the slowest-executing process. This capability makes MacPitts well-suited for design of controller-oriented chips.

The chip design process with MacPitts can be understood initially as algorithmic optimization. The test algorithm is written, tested in the interpreter, and compiled to cif. Then an expanded version of the test algorithm is written and tested in the interpreter. The expanded version is compiled to cif, a circuit extraction is made, and the electrical characteristics and speed of the chip are determined. Alternate solutions are then considered, and tested in the same fashion. The best of these is chosen as the archetype for the desired chip. The archetype must have sufficiently few signals, ports, registers, and flags to permit testing in the interpreter (a maximum of 36). The algorithm is then expanded again to cover the desired chip function. The final algorithm is compiled to cif, a circuit extraction is made, and then the chip is tested electrically. If there are too many variables to permit command interpreter display, the algorithm is tested with a switch-level simulator (this exercises both the algorithm and the circuit). Further analyses with a power estimator and a timing analyzer are done to see that the chip operates within specifications. If the chip operates too slow, parallelism should be applied to the algorithm where possible, in an attempt to trade speed for silicon area.

B. RECOMMENDATIONS

This thesis also investigated a number of MacPitts errors and shortcomings. The following recommendations should be considered:

1. Have the the light controller chips fabricated by MOSIS for testing at Naval Postgraduate school, and compare with the results from Crystal.

2. The Weinberger array errors as depicted in Chapter II are thought to result from incorrect installation of MacPitts under Unix 4.2. It would be fruitful to search for a Unix-dependent roundoff error in the instantiation of partial-gate-input-ground-right and partial-gate-input-ground-left. The poly interconnections between data and control also suffer a lateral displacement/gap error, and

the solution to the partial gate problem is likely to solve this one also. Similar errors were also noted in the data path, usually between vertical metal lines and horizontal Vdd/GND busses.

3. New Mead-Conway organelles (cf. Chapter III) should be tried as replacements for the MacPitts data path organelles. This will require comparison between similar structures with Powest and Crystal, and selection of the better circuit. MacPitts will connect the new organelles properly if the pitch is preserved.

4. The error of shorted flag traces occurs almost every time a flag is declared. The vertical flag lines intersect the horizontal clock traces at a via cut, which shorts the flag signal and does not permit it to pass to control. The solution to this error is best solved by a conditional test in the routing algorithm. If the flag traces run close to the Vdd/ground comb, then the traces must be moved in towards the center of the chip.

5. The possibility of replacing the slow Weinberger array with a PLA should be considered. This solution will entail a complete rewrite of the control.lisp source file, and major modification to other files which depend on or interact with control.lisp. A study of plague and plagen (or eqntott and tpla) is the best place to start, with a view towards replacing the Weinberger array with a compact PLA. The difficulty will lie in the interface between the PLA logic equation specification (in plague or eqntott) and the MacPitts algorithmic language.

6. The problem of vestigal instantiation (sequencers, unconnected vertical poly runs from the data path) could be solved with a simple test using list processing primitives. If the organelles or wires are not needed, then skip the instantiation process.

7. The problem of the unconnected Vdd bus only occurs in very small chips, but should be simple to correct. A metal routing up and to the left, to

connect to the Vdd comb is required. The simple solution is to explicitly specify a connecting wire in the CLL-like language used in the MacPitts source code. The more instructive solution is to write the Franz LISP code to decide if a jumper wire is needed, and if so, to create one.

8. A menu invoking Crystal, Esim, Powest, and Mextra would speed up the design cycle. The menu could be incorporated in MacPitts, but would probably be just as good external to MacPitts. A timing analysis is necessary in the compilation of the chip, however. If it had existed during the Hamming 15/4 error corrector example (Chapter VI), the choice of an archetype chip would have been simpler.

9. The VT-100 terminal screen is too small to display the interpreter session of all the signals, flags, registers, and ports which occur on even a moderate-sized MacPitts chip. A windowing capability is needed. The source file interpret.lisp contains the command interpreter logic. The interpreter is functionally a dynamic debugger, similar to those in CP/M or VMS (but without the ability to change the source code). The interpreter has a very slow response time to terminal inputs for all but the simplest chip algorithms, and it would be useful to speed it up also if other modifications are planned.

10. SPICE would be a valuable addition to timing analysis. Currently, SPICE 2g6 is not installed on the VAX-11/780 at Naval Postgraduate School. A plot of the SPICE output is also desired, but not available under the currently installed version of Unix 4.2.

11. The capability to scale the MacPitts designs to sizes other than multiples of 200 or 250 centimicrons is needed for future applications. The ability to scale in multiples of 25 centimicrons is suggested, where the designer chooses the option at compile time in the MacPitts <options> field.

12. MacPitts currently places pads on only three sides of the chip frame. A better design would permit

pads to be placed on all four sides of the chip. This would also allow faster chips, due to shortened inter-chip wires.

13. The capability of automatic test vector generation and evaluation is lacking. The command interpreter should be able to access an existing file for testing and write the results of the tests to another file.

14. The ability to display transistor density as one of the compiler statistics should be incorporated. This would be a simple task, since MacPitts already computes the chip dimensions and the number of transistors, and writes each of these values to the statistics output file.

15. A serial implementation of the Hamming 15/4 error detector/ corrector should be attempted using primitive polynomials [Ref. 13], [Ref. 5:pp. 200]. The throughput should be compared to the parallel 15/4 error corrector. The interesting problem is to solve the differing bandwidths at the input and output of the shift register. MacPitts may not be able to cope with this requirement, and will likely be slower than the parallel architecture (in the throughput sense) regardless.

16. A MacPitts prototype FIR or IIR digital filter should be attempted. The first model should be an FIR four-bit prototype, and this algorithm can then be expanded to the floating point version of larger word length. An excellent reference for the designer is [Ref. 14:pp. 541], where the algorithmic aspects of digital filter design are explained.

17. Faster graphics are required for the VLSI graphics terminal (Caesar). A better (i.e., quicker) terminal should be considered.

18. The Backus-Naur file (BNF) included with the MacPitts source code specifies allowed algorithmic syntax. The macro and lambda forms should be investigated with a view to incorporating macros into the algorithms.

19. It would speed up the design time and confer added versatility on MacPitts if the input port width could be specified as a variable. The word lengths would then be assigned according to another single statement in the MacPitts algorithm. For instance

```
(def face port input (*))
(def data_word_width 16)
```

would assign a 16-bit width to the variable <face>, and to any other occurrences of the asterisk.

APPENDIX A

CHAPTER III LISTINGS

```
(((destination z)
  (source a)
  (source b)
  (source c)
  (source d)
  (source e)
  (logo fivand)
  (word-length 1)
  (ground 1)
  (port a input (2))
  (port b input (3))
  (port c input (4))
  (port d input (5))
  (port e input (6))
  (port z output (7))
  (phia 8)
  (phib 9)
  (phic 10)
  (power 11))
 nil
 ((organelle and -1 (((port-input d) (port-input e))))
  (organelle and -2 (((port-input c) (internal 1))))
  (organelle and -3 (((port-input b) (internal 2))))
  (organelle and -4 (((port-input a) (internal 3))))
  (port-output z (((internal 4)))))
 nil
 ((10 (phic))
  (9 (phib))
  (8 (phia))
  (1 (ground))
  (11 (power))
  (2 (input (a 0) (port-input a 0)))
  (3 (input (b 0) (port-input b 0)))
  (4 (input (c 0) (port-input c 0)))
  (5 (input (d 0) (port-input d 0)))
  (6 (input (e 0) (port-input e 0)))
  (7 (output8 (z 0) (port-output z 0)))))
```

Data Path Five Input AND Gate .obj File

```
Statistic - for project fivand
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 68, 58 - Reading source file - fivand.mac
Herald - 72, 58 - Reading library from - /vlsi/macpit/library
Herald - 901, 611 - Processing definitions
Herald - 903, 611 - Evaluating evals
Herald - 986, 611 - Expanding macros
Herald - 989, 611 - Extracting sources
Herald - 990, 611 - Extracting destinations
Herald - 991, 611 - Extracting labels
Herald - 991, 611 - Extracting sequencers
Herald - 991, 611 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 0
Statistic - Number of gates is 0
Statistic - Data-path has 5 Units
Herald - 1383, 901 - Outputing .obj file
Herald - 1413, 901 - Extruding gates
Statistic - Control has 0 columns
Herald - 1516, 997 - Extruding straps
Statistic - Circuit has 98 transistors
Statistic - Control has 0 tracks
Statistic - Power consumption is 0.038114 Watts
Herald - 1679, 1095 - Laying out data-path
Herald - 1815, 1192 - Organelle unit# 1 bit 0
Herald - 2014, 1290 - Organelle unit# 2 bit 0
Herald - 2168, 1391 - Organelle unit# 3 bit 0
Herald - 2332, 1498 - Organelle unit# 4 bit 0
Herald - 2385, 1498 - Organelle unit# 5 bit 0
Statistic - Data-path internal bus uses 6 tracks
Herald - 2539, 1600 - Laying out control
Herald - 2542, 1600 - Laying out flags
Herald - 2543, 1600 - Laying out river
Herald - 2545, 1600 - Laying out wing
Herald - 2547, 1600 - Laying out skeleton
Herald - 2683, 1699 - Laying out pins
Statistic - Dimensions are 1.805000 mm by 1.872500 mm
Herald - 5299, 3105 - Outputing .cif file
Statistic - Memory used - 357K
Statistic - Compilation took 1.534722 CPU minutes
Statistic - Garbage collection took 0.893333 CPU minutes
Statistic - For a total of 33 garbage collections
```

Script of Compilation of Data Path Five Input AND Gate

```
94 41 64200 79400;
94 42 82200 79400;
94 43 100200 79400;
94 a 46300 79600;
94 41 64300 79600;
94 42 82300 79600;
94 43 100300 79600;
94 54 48000 79900;
94 41 54200 79900;
94 55 66000 79900;
94 42 72200 79900;
94 56 84000 79900;
94 43 90200 79900;
94 57 102000 79900;
94 z 108200 79900;
94 54 49800 80400;
94 41 55500 80400;
94 55 67800 80400;
94 42 73500 80400;
94 56 85800 80400;
94 43 91500 80400;
94 57 103800 80400;
94 z 109500 80400;
94 a 46300 80400;
94 41 64300 80400;
94 42 82300 80400;
94 43 100300 80400;
94 Vdd 52000 80600;
94 Vdd 57700 80600;
94 Vdd 70000 80600;
94 Vdd 75700 80600;
94 Vdd 88000 80600;
94 Vdd 93700 80600;
94 Vdd 106000 80600;
94 Vdd 111700 80600;
94 54 49800 81600;
94 41 55500 81600;
94 55 67800 81600;
94 42 73500 81600;
94 56 85800 81600;
94 43 91500 81600;
94 57 103800 81600;
94 z 109500 81600;
94 54 49800 82400;
94 41 55500 82400;
94 55 67800 82400;
94 42 73500 82400;
94 56 85800 82400;
94 43 91500 82400;
94 57 103800 82400;
94 z 109500 82400;
94 z 116500 83600;
94 e 97200 84900;
94 z 109400 84900;
94 z 113500 84900;
94 d 79200 86100;
94 43 91400 86100;
94 43 95500 86100;
94 GND 41500 71700;
94 Vdd 52000 76800;
94 Vdd 57700 76800;
94 Vdd 70000 76800;
94 Vdd 75700 76800;
94 Vdd 88000 76800;
94 Vdd 93700 76800;
94 Vdd 106000 76800;
94 Vdd 111700 76800;
94 b 43200 76900;
94 c 61200 76900;
94 d 79200 76900;
94 e 97200 76900;
94 z 113500 76900;
94 b 43200 76900;
94 GND 48000 76900;
94 c 61200 76900;
94 GND 66000 76900;
94 d 79200 76900;
94 GND 84000 76900;
94 e 97200 76900;
94 GND 102000 76900;
94 z 113500 76900;
94 b 46300 77100;
94 c 64300 77100;
94 d 82300 77100;
94 e 100300 77100;
94 b 45000 77100;
94 c 63000 77100;
94 d 81000 77100;
94 e 99000 77100;
94 b 46300 77800;
94 c 64300 77800;
94 d 82300 77800;
94 e 100300 77800;
94 GND 53700 78100;
94 GND 71700 78100;
94 GND 89700 78100;
94 GND 107700 78100;
94 a 41500 78600;
94 41 59500 78600;
94 42 77500 78600;
94 43 95500 78600;
94 a 41500 78600;
94 45 48000 78600;
94 41 59500 78600;
94 47 66000 78600;
94 42 77500 78600;
94 49 84000 78600;
94 43 95500 78600;
94 51 102000 78600;
94 z 116500 78900;
94 z 116500 78900;
94 54 53200 79300;
94 55 71200 79300;
94 56 89200 79300;
94 57 107200 79300;
94 a 46200 79400;
94 c 61200 87400;
94 42 73400 87400;
94 42 77500 87400;
94 b 43200 88600;
94 41 55400 88600;
94 41 59500 88600;
94 a 41500 89900;
```

Data Path Five Input AND Gate .nodes File

```
Crystal, v.2
: build 5andcr.sim
[Ø:ØØ.1u Ø:ØØ.2s 21k]
: inputs a b c d e
[Ø:ØØ.Øu Ø:ØØ.Øs 3Øk]
: outputs z
[Ø:ØØ.Øu Ø:ØØ.Øs 3Øk]
: delay a -1 Ø
Marking transistor flow...
Setting Vdd to 1...
Setting GND to Ø...
(9 stages examined.)
[Ø:ØØ.1u Ø:ØØ.1s 31k]
: delay b -1 Ø
(1 stages examined.)
[Ø:ØØ.Øu Ø:ØØ.Øs 31k]
: delay c -1 Ø
(1 stages examined.)
[Ø:ØØ.Øu Ø:ØØ.Øs 31k]
: delay d -1 Ø
(1 stages examined.)
[Ø:ØØ.Øu Ø:ØØ.Øs 31k]
: delay e -1 Ø
(1 stages examined.)
[Ø:ØØ.Øu Ø:ØØ.Øs 31k]
: critical
Node z is driven low at 86.Ø1ns
      ...through fet at (541, [illegible]) to GND after
    57 is driven high at 7Ø.55ns
      ...through fet at (519, 4Ø5) to Vdd after
    43 is driven low at 61.39ns
      ...through fet at (451, 397) to GND after
    56 is driven high at 5Ø.4Øns
      ...through fet at (429, 4Ø5) to Vdd after
    42 is driven low at 41.22ns
      ...through fet at (361, 397) to GND after
    55 is driven high at 29.99ns
      ...through fet at (339, 4Ø5) to Vdd after
    41 is driven low at 2Ø.81ns
      ...through fet at (271, 397) to GND after
    54 is driven high at 9.4Øns
      ...through fet at (249, 4Ø5) to Vdd after
    a is driven low at Ø.ØØns
[Ø:ØØ.1u Ø:ØØ.1s 31k]
: critical -g 5andcr.dum
[Ø:ØØ.1u Ø:ØØ.1s 36k]
: quit
[Ø:ØØ.4u Ø:ØØ.4s 36k] Crystal done.
% ^D
```

Data Path Five Input AND Crystal Session

```
vis +1
push 541 397 2 2
paint e
label [8]86.Øns,fall
push 519 4Ø5 2 2
paint e
label [7]7Ø.6ns,rise
push 451 397 2 2
paint e
label [6]61.4ns,fall
push 429 4Ø5 2 2
paint e
label [5]5Ø.4ns,rise
push 361 397 2 2
paint e
label [4]41.2ns,fall
push 339 4Ø5 2 2
paint e
label [3]3Ø.Øns,rise
push 271 397 2 2
paint e
label [2]2Ø.8ns,fall
push 249 4Ø5 2 2
paint e
label [1]9.4ns,rise
```

Data Path Five Input AND Critical Nodes

```
% powest -p <a5andcr.sim
gamma=Ø.4V**.5, tox=9e-Ø8m, uØ=Ø.Ø8m**2/V-s
vdd=5V, vtd=-3.5V, vte=Ø.8V, vsb=2V
#devs   Pdc_avg (W)      Pdc_max (W)      type

Ø       Ø.ØØØØØØ         Ø.ØØØØØØ         enhancement pullups
8       Ø.ØØØ94Ø         Ø.ØØ1879         depletion pullups
Ø       Ø.ØØØØØØ         Ø.ØØØØØØ         special depletion pullups

8       Ø.ØØØ94Ø         Ø.ØØ1879         TOTAL
% ^D
```

Data Path Five Input AND Powest Analysis

```
(((destination z)
  (source a)
  (source b)
  (source c)
  (source d)
  (source e)
  (logo fiveand)
  (word-length 1)
  (ground 1)
  (signal a input 5)
  (signal b input 6)
  (signal c input 7)
  (signal d input 8)
  (signal e input 9)
  (signal z output 1Ø)
  (phia 2)
  (phib 3)
  (phic 4)
  (power 11))
 nil
 nil
 (((signal-output z) (nor ((primitive (gate 1Ø)))))
  ((gate 1Ø)
   (nor
    ((primitive (gate 9))
     (primitive (gate 8))
     (primitive (gate 7))
     (primitive (gate 6))
     (primitive (gate 5)))))
  ((gate 9)
   (nor
    ((primitive (gate 4))
     (primitive (gate 3))
     (primitive (gate 2))
     (primitive (gate 1))
     (primitive (gate Ø))
     (primitive (signal-input a))
     (primitive (signal-input b))
     (primitive (signal-input c))
     (primitive (signal-input d)))))
  ((gate 8)
   (nor
    ((primitive (gate 4))
     (primitive (gate 3))
     (primitive (gate 2))
     (primitive (gate 1))
     (primitive (gate Ø))
     (primitive (signal-input a))
     (primitive (signal-input b))
     (primitive (signal-input c)))))
  ((gate 7)
   (nor
    ((primitive (gate 4))
     (primitive (gate 3))
     (primitive (gate 2))
     (primitive (gate 1))
     (primitive (gate Ø))
```

Control Path Five Input AND .obj File

```
      (primitive (signal-input a))
      (primitive (signal-input b)))))
   ((gate 6)
    (nor
     ((primitive (gate 4))
      (primitive (gate 3))
      (primitive (gate 2))
      (primitive (gate 1))
      (primitive (gate Ø))
      (primitive (signal-input a)))))
   ((gate 5)
    (nor
     ((primitive (gate 4))
      (primitive (gate 3))
      (primitive (gate 2))
      (primitive (gate 1))
      (primitive (gate Ø)))))
   ((gate 4) (nor ((primitive (signal-input a)))))
   ((gate 3) (nor ((primitive (signal-input b)))))
   ((gate 2) (nor ((primitive (signal-input c)))))
   ((gate 1) (nor ((primitive (signal-input d)))))
   ((gate Ø) (nor ((primitive (signal-input e))))))
  ((4 (phic))
   (3 (phib))
   (2 (phia))
   (1 (ground))
   (11 (power))
   (9 (input e (signal-input e)))
   (8 (input d (signal-input d)))
   (7 (input c (signal-input c)))
   (6 (input b (signal-input b)))
   (5 (input a (signal-input a)))
   (1Ø (output8 z (signal-output z)))))
```

Control Path Five Input AND .obj File (continued)

```
Script started on Mon Apr 15 22:29:07 1985
% macpitts fiveand.herald
Statistic - for project fiveand
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 63, 55 - Reading source file - fiveand.mac
Herald - 70, 55 - Reading library from - /vlsi/macpit/library
Herald - 896, 604 - Processing definitions
Herald - 898, 604 - Evaluating evals
Herald - 983, 604 - Expanding macros
Herald - 1103, 701 - Extracting sources
Herald - 1108, 701 - Extracting destinations
Herald - 1110, 701 - Extracting labels
Herald - 1110, 701 - Extracting sequencers
Herald - 1111, 701 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 4
Statistic - Number of gates is 12
Statistic - Data-path has 0 Units
Herald - 1946, 1286 - Outputing .obj file
Herald - 2002, 1286 - Extruding gates
Statistic - Control has 17 columns
Herald - 4001, 2417 - Extruding straps
Statistic - Circuit has 136 transistors
Statistic - Control has 11 tracks
Statistic - Power consumption is 0.040723 Watts
Herald - 4183, 2517 - Laying out data-path
Statistic - Data-path internal bus uses 0 tracks
Herald - 4186, 2517 - Laying out control
Herald - 4997, 2943 - Laying out flags
Herald - 4999, 2943 - Laying out river
Herald - 5000, 2943 - Laying out wing
Herald - 5018, 2943 - Laying out skeleton
Herald - 5054, 2943 - Laying out pins
Statistic - Dimensions are 1.772500 mm by 1.905000 mm
Herald - 7361, 4042 - Outputing .cif file
Statistic - Memory used - 349K
Statistic - Compilation took 2.106111 CPU minutes
Statistic - Garbage collection took 1.153889 CPU minutes
Statistic - For a total of 41 garbage collections
% ^D
script done on Mon Apr 15 22:34:42 1985
```

Control Path Five Input AND Gate Script File

```
94 Vdd 41000 46000;
94 Vdd 45200 47700;
94 Vdd 48200 47700;
94 Vdd 50400 47700;
94 Vdd 53400 47700;
94 Vdd 55700 47700;
94 Vdd 58700 47700;
94 Vdd 62700 47700;
94 Vdd 65700 47700;
94 Vdd 67900 47700;
94 Vdd 73200 47700;
94 Vdd 78400 47700;
94 Vdd 81400 47700;
94 14 45000 48900;
94 15 48000 48900;
94 16 50200 48900;
94 z 53200 48900;
94 18 55500 48900;
94 19 58500 48900;
94 20 62500 48900;
94 21 65500 48900;
94 22 67700 48900;
94 23 73000 48900;
94 24 78200 48900;
94 25 81200 48900;
94 14 45500 51200;
94 15 48500 51200;
94 16 50700 51200;
94 z 53700 51200;
94 18 56000 51200;
94 19 59000 51200;
94 20 63000 51200;
94 21 66000 51200;
94 22 68200 51200;
94 23 73500 51200;
94 24 78700 51200;
94 25 81700 51200;
94 14 45000 52900;
94 15 48500 52900;
94 16 50200 52900;
94 z 53700 52900;
94 18 55500 52900;
94 19 59000 52900;
94 20 62500 52900;
94 21 66000 52900;
94 22 67700 52900;
94 23 73000 52900;
94 24 78200 52900;
94 25 81700 52900;
94 d 71200 53900;
94 22 67700 53900;
94 d 71200 53900;
94 GND 48500 54700;
94 GND 78700 54700;
94 GND 81700 54700;
94 GND 46700 54900;
94 GND 52000 54900;
94 GND 57200 54900;
94 GND 64200 54900;
94 GND 69500 54900;
94 GND 80000 54900;
94 GND 46700 54900;
94 GND 52000 54900;
94 GND 57200 54900;
94 GND 64200 54900;
94 GND 69500 54900;
94 GND 80000 54900;
94 15 48500 55900;
94 25 81700 55900;
94 z 53700 56700;
94 18 56000 56700;
94 19 59000 56700;
94 20 63000 56700;
94 22 68200 56700;
94 24 78700 56700;
94 16 50200 57900;
94 GND 56000 58700;
94 GND 59000 58700;
94 GND 63000 58700;
94 GND 68200 58700;
94 GND 78700 58700;
94 GND 52000 58900;
94 GND 57200 58900;
94 GND 64200 58900;
94 GND 69500 58900;
94 GND 80000 58900;
94 a 41500 59900;
94 a 41500 59900;
94 23 73000 59900;
94 16 50700 60700;
94 18 56000 60700;
94 19 59000 60700;
94 20 63000 60700;
94 22 68200 60700;
94 24 78700 60700;
94 14 45000 61900;
94 GND 56000 62700;
94 GND 59000 62700;
94 GND 63000 62700;
94 GND 68200 62700;
94 GND 78700 62700;
94 GND 46700 62900;
94 GND 57200 62900;
94 GND 64200 62900;
94 GND 69500 62900;
94 GND 80000 62900;
94 b 43200 63900;
94 b 43200 63900;
94 14 45500 64700;
94 18 56000 64700;
94 20 63000 64700;
94 22 68200 64700:
94 24 78700 64700;
94 19 59000 65000;
94 21 66000 65900;
94 GND 56000 66700;
94 GND 59000 66700;
94 GND 63000 66700;
94 GND 68200 66700;
94 GND 78700 66700;
94 GND 74700 66900;
94 GND 46700 66900;
94 GND 57200 66900;
94 GND 64200 66900;
94 GND 69500 66900;
94 GND 74700 66900;
94 GND 80000 66900;
94 e 76500 67900;
94 19 59000 67900;
94 e 76500 67900;
94 15 48500 68700;
94 20 63000 68700;
94 22 68200 68700;
94 23 73500 68700;
94 24 78700 68700;
94 21 66000 69000;
94 c 60700 69900;
94 18 55500 69300;
94 c 60700 69900;
94 GND 48500 70700;
94 GND 63000 70700;
94 GND 66000 70700;
94 GND 78700 70700;
94 GND 46700 70900;
94 GND 64200 70300;
94 GND 80000 70900;
94 24 78200 71900;
94 15 48500 72700;
94 20 62500 73900;
94 GND 48500 74700;
94 GND 46700 74900;
94 a 41500 75900;
94 b 43200 75900;
94 z 53700 75900;
94 c 60700 75900;
94 d 71200 75900;
94 e 76500 75900;
```

Control Path Five Input AND Gate .nodes File

APPENDIX B

CHAPTER IV LISTINGS

```
Statistic - for project gc
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 64, 57 - Reading source file - gc.mac
Herald - 69, 57 - Reading library from - /vlsi/macpit/library
Herald - 911, 622 - Processing definitions
Herald - 912, 622 - Evaluating evals
Herald - 996, 622 - Expanding macros
Herald - 1009, 622 - Extracting sources
Herald - 1012, 622 - Extracting destinations
Herald - 1108, 716 - Extracting labels
Herald - 1108, 716 - Extracting sequencers
Herald - 1110, 716 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 4
Statistic - Number of gates is 26
Statistic - Data-path has 7 Units
Herald - 2625, 1722 - Outputing .obj file
Herald - 2716, 1722 - Extruding gates
Statistic - Control has 31 columns
Herald - 8491, 4785 - Extruding straps
Statistic - Circuit has 280 transistors
Statistic - Control has 12 tracks
Statistic - Power consumption is 0.0559l0 Watts
Herald - 8910, 4993 - Laying out data-path
Herald - 9070, 5099 - Organelle unit# 1 bit 1
Herald - 9263, 5207 - Organelle unit# 1 bit 0
Herald - 9318, 5207 - Organelle unit# 2 bit 1
Herald - 9549, 5313 - Organelle unit# 2 bit 0
Herald - 9636, 5313 - Organelle unit# 3 bit 1
Herald - 9784, 5421 - Organelle unit# 3 bit 0
Herald - 9846, 5421 - Organelle unit# 4 bit 1
Herald - 10274, 5652 - Organelle unit# 4 bit 0
Herald - 10470, 5765 - Organelle unit# 5 bit 1
Herald - 10509, 5765 - Organelle unit# 5 bit 0
Herald - 10578, 5765 - Organelle unit# 6 bit 1
Herald - 10801, 5876 - Organelle unit# 6 bit 0
Herald - 10997, 5989 - Organelle unit# 7 bit 1
Herald - 11014, 5989 - Organelle unit# 7 bit 0
Statistic - Data-path internal bus uses 3 tracks
Herald - 11096, 5989 - Laying out control
Herald - 13020, 6925 - Laying out flags
Herald - 13023, 6925 - Laying out river
Herald - 13168, 7041 - Laying out wing
Herald - 13177, 7041 - Laying out skeleton
Herald - 13262, 7041 - Laying out pins
Statistic - Dimensions are 2.587500 mm by 1.982500 mm
Herald - 15882, 8254 - Outputing .cif file
Statistic - Memory used - 403K
Statistic - Compilation took 4.487778 CPU minutes
Statistic - Garbage collection took 2.328889 CPU minutes
Statistic - For a total of 79 garbage collections
```

GC.script

```
Statistic - for project gc2
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 61, 54 - Reading source file - gc2.mac
Herald - 64, 54 - Reading library from - /vlsi/macpit/library
Herald - 882, 596 - Processing definitions
Herald - 884, 596 - Evaluating evals
Herald - 967, 596 - Expanding macros
Herald - 986, 596 - Extracting sources
Herald - 1084, 692 - Extracting destinations
Herald - 1086, 692 - Extracting labels
Herald - 1087, 692 - Extracting sequencers
Herald - 1090, 692 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 4
Statistic - Number of gates is 27
Statistic - Data-path has 8 Units
Herald - 2661, 1695 - Outputing .obj file
Herald - 2766, 1695 - Extruding gates
Statistic - Control has 32 columns
Herald - 9213, 5045 - Extruding straps
Statistic - Circuit has 288 transistors
Statistic - Control has 13 tracks
Statistic - Power consumption is 0.057477 Watts
Herald - 9651, 5249 - Laying out data-path
Herald - 9822, 5356 - Organelle unit# 1 bit 1
Herald - 10022, 5464 - Organelle unit# 1 bit 0
Herald - 10072, 5464 - Organelle unit# 2 bit 1
Herald - 10114, 5464 - Organelle unit# 2 bit 0
Herald - 10270, 5571 - Organelle unit# 3 bit 1
Herald - 10503, 5684 - Organelle unit# 3 bit 0
Herald - 10585, 5684 - Organelle unit# 4 bit 1
Herald - 10718, 5792 - Organelle unit# 4 bit 0
Herald - 10755, 5792 - Organelle unit# 5 bit 1
Herald - 11169, 6017 - Organelle unit# 5 bit 0
Herald - 11254, 6017 - Organelle unit# 6 bit 1
Herald - 11422, 6128 - Organelle unit# 6 bit 0
Herald - 11494, 6128 - Organelle unit# 7 bit 1
Herald - 11723, 6241 - Organelle unit# 7 bit 0
Herald - 11916, 6353 - Organelle unit# 8 bit 1
Herald - 11936, 6353 - Organelle unit# 8 bit 0
Statistic - Data-path internal bus uses 3 tracks
Herald - 12034, 6353 - Laying out control
Herald - 14219, 7417 - Laying out flags
Herald - 14224, 7417 - Laying out river
Herald - 14374, 7534 - Laying out wing
Herald - 14383, 7534 - Laying out skeleton
Herald - 14478, 7534 - Laying out pins
Statistic - Dimensions are 2.687500 mm by 1.982500 mm
Herald - 17205, 8788 - Outputing .cif file
Statistic - Memory used - 408K
Statistic - Compilation took 4.823334 CPU minutes
Statistic - Garbage collection took 2.441111 CPU minutes
Statistic - For a total of 83 garbage collections
```

Gc2.scr

```
Statistic - for project stop
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 63, 56 - Reading source file - stop.mac
Herald - 74, 56 - Reading library from - /vlsi/macpit/library
Herald - 877, 588 - Processing definitions
Herald - 878, 588 - Evaluating evals
Herald - 961, 588 - Expanding macros
Herald - 1088, 681 - Extracting sources
Herald - 1094, 681 - Extracting destinations
Herald - 1102, 681 - Extracting labels
Herald - 1102, 681 - Extracting sequencers
Herald - 1107, 681 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 5
Statistic - Number of gates is 37
Statistic - Data-path has 3 Units
Herald - 2983, 1885 - Outputing .obj file
Herald - 3104, 1885 - Extruding gates
Statistic - Control has 43 columns
Herald - 17705, 9477 - Extruding straps
Statistic - Circuit has 268 transistors
Statistic - Control has 14 tracks
Statistic - Power consumption is 0.054698 Watts
Herald - 18256, 9790 - Laying out data-path
Herald - 18279, 9790 - Organelle unit# 1 bit 1
Herald - 18773, 10113 - Organelle unit# 1 bit 0
Herald-- 18830, 10113 - Organelle unit# 2 bit 1
Herald - 19001, 10220 - Organelle unit# 2 bit 0
Herald - 19075, 10220 - Organelle unit# 3 bit 1
Herald - 19091, 10220 - Organelle unit# 3 bit 0
Statistic - Data-path internal bus uses 2 tracks
Herald - 19244, 10327 - Laying out control
Herald - 21284, 11356 - Laying out flags
Herald - 21286, 11356 - Laying out river
Herald - 21307, 11356 - Laying out wing
Herald - 21333, 11356 - Laying out skeleton
Herald - 21382, 11356 - Laying out pins
Statistic - Dimensions are 2.107500 mm by 2.207500 mm
Herald - 24464, 12791 - Outputing .cif file
Statistic - Memory used - 403K
Statistic - Compilation took 6.877223 CPU minutes
Statistic - Garbage collection took 3.587222 CPU minutes
Statistic - For a total of 123 garbage collections
```

Lc2.scr

```
Statistic - for project b5
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 65, 53 - Reading source file - b4.mac
Herald - 74, 53 - Reading library from - /vlsi/macpit/library
Herald - 898, 596 - Processing definitions
Herald - 899, 596 - Evaluating evals
Herald - 980, 596 - Expanding macros
Herald - 1106, 686 - Extracting sources
Herald - 1113, 686 - Extracting destinations
Herald - 1118, 686 - Extracting labels
Herald - 1118, 686 - Extracting sequencers
Herald - 1121, 686 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 4
Statistic - Number of gates is 53
Statistic - Data-path has 10 Units
Herald - 4032, 2550 - Outputing .obj file
Herald - 4243, 2550 - Extruding gates
Statistic - Control has 63 columns
Herald - 25458, 12382 - Extruding straps
Statistic - Circuit has 1208 transistors
Statistic - Control has 27 tracks
Statistic - Power consumption is 0.201805 Watts
Herald - 26808, 13048 - Laying out data-path
Herald - 27264, 13272 - Organelle unit# 1 bit 4
Herald - 27788, 13612 - Organelle unit# 1 bit 3
Herald - 27815, 13612 - Organelle unit# 1 bit 2
Herald - 27841, 13612 - Organelle unit# 1 bit 1
Herald - 27983, 13727 - Organelle unit# 1 bit 0
Herald - 28111, 13727 - Organelle unit# 2 bit 4
Herald - 28292, 13845 - Organelle unit# 2 bit 3
Herald - 28320, 13845 - Organelle unit# 2 bit 2
Herald - 28349, 13845 - Organelle unit# 2 bit 1
Herald - 28499, 13965 - Organelle unit# 2 bit 0
Herald - 28634, 13965 - Organelle unit# 3 bit 4
Herald - 28886, 14082 - Organelle unit# 3 bit 3
Herald - 28920, 14082 - Organelle unit# 3 bit 2
Herald - 29186, 14313 - Organelle unit# 3 bit 1
Herald - 29220, 14313 - Organelle unit# 3 bit 0
Herald - 29360, 14313 - Organelle unit# 4 bit 4
Herald - 29497, 14430 - Organelle unit# 4 bit 3
Herald - 29509, 14430 - Organelle unit# 4 bit 2
Herald - 29521, 14430 - Organelle unit# 4 bit 1
Herald - 29532, 14430 - Organelle unit# 4 bit 0
Herald - 29602, 14430 - Organelle unit# 5 bit 4
Herald - 30093, 14671 - Organelle unit# 5 bit 3
Herald - 30290, 14794 - Organelle unit# 5 bit 2
Herald - 30358, 14794 - Organelle unit# 5 bit 1
Herald - 30551, 14919 - Organelle unit# 5 bit 0
Herald - 31072, 15171 - Organelle unit# 6 bit 4
Herald - 31346, 15296 - Organelle unit# 6 bit 3
Herald - 31388, 15296 - Organelle unit# 6 bit 2
Herald - 31431, 15296 - Organelle unit# 6 bit 1
Herald - 31599, 15421 - Organelle unit# 6 bit 0
Herald - 31766, 15421 - Organelle unit# 7 bit 4
Herald - 31972, 15545 - Organelle unit# 7 bit 3
Herald - 32001, 15545 - Organelle unit# 7 bit 2
Herald - 32031, 15545 - Organelle unit# 7 bit 1
Herald - 32188, 15671 - Organelle unit# 7 bit 0
```

B5.scr

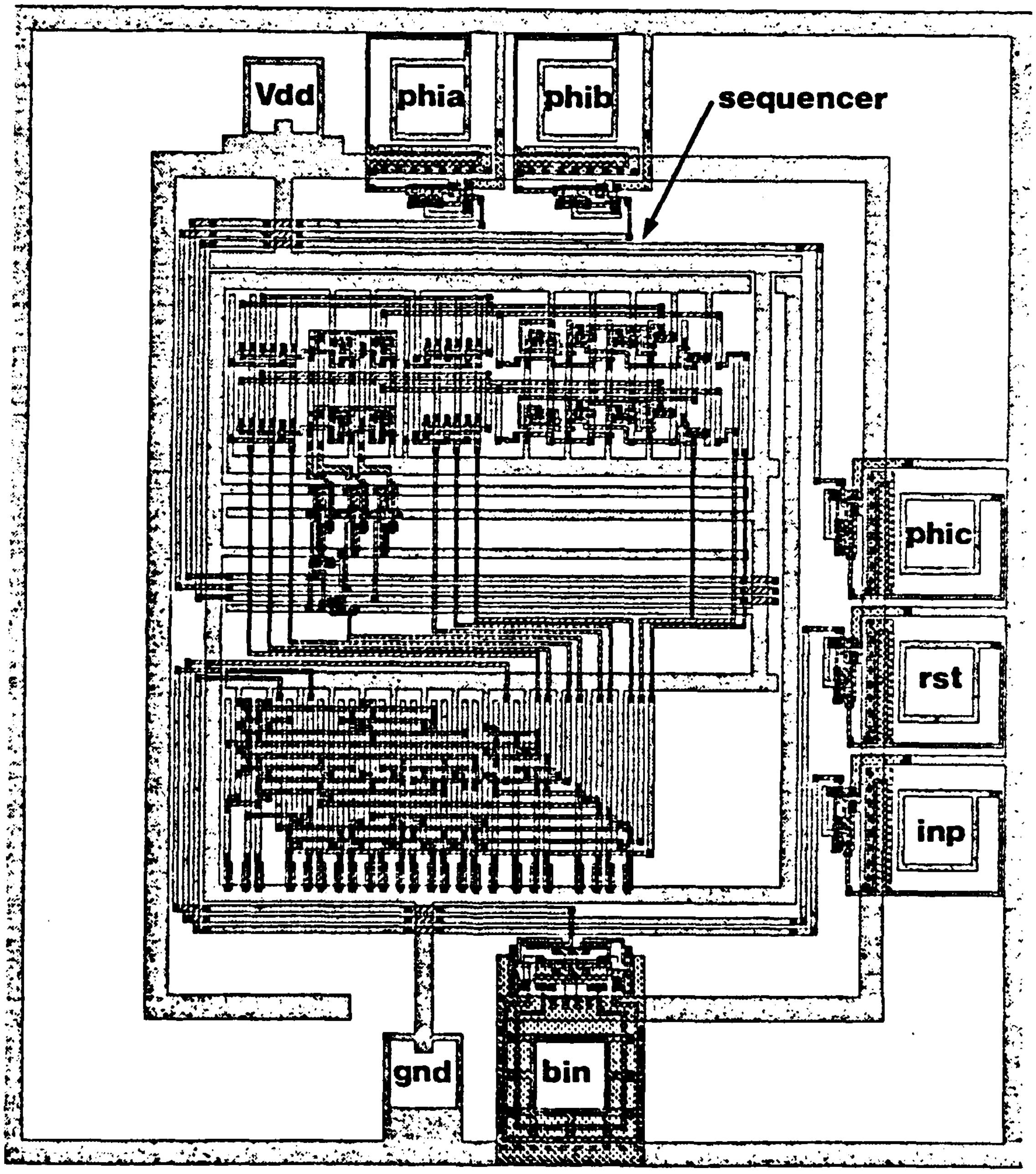
```
Herald - 32331, 15671 - Organelle unit# 8 bit 4
Herald - 32342, 15671 - Organelle unit# 8 bit 3
Herald - 32354, 15671 - Organelle unit# 8 bit 2
Herald - 32493, 15800 - Organelle unit# 8 bit 1
Herald - 32505, 15800 - Organelle unit# 8 bit 0
Herald - 32560, 15800 - Organelle unit# 9 bit 4
Herald - 32916, 15930 - Organelle unit# 9 bit 3
Herald - 33125, 16060 - Organelle unit# 9 bit 2
Herald - 33341, 16196 - Organelle unit# 9 bit 1
Herald - 33422, 16196 - Organelle unit# 9 bit 0
Herald - 33983, 16459 - Organelle unit# 10 bit 4
Herald - 34082, 16459 - Organelle unit# 10 bit 3
Herald - 34297, 16590 - Organelle unit# 10 bit 2
Herald - 34515, 16722 - Organelle umit# 10 bit 1
Herald - 34601, 16722 - Organelle unit# 10 bit 0
Statistic - Data-path internal bus uses 5 tracks
Herald - 35348, 16992 - Laying out control
Herald - 41246, 19921 - Laying out flags
Herald - 41742, 20059 - Laying out river
Herald - 41993, 20197 - Laying out wing
Herald - 42015, 20197 - Laying out skeleton
Herald - 42180, 20197 - Laying out pins
Statistic - Dimensions are 5.770000 mm by 3.125000 mm
Herald - 49229, 23494 - Outputing .cif file
Statistic - Memory used - 518K
Statistic - Compilation took 13.804167 CPU minutes
Statistic - Garbage collection took 6.569723 CPU minutes
Statistic - For a total of 199 garbage collections
```

B5.scr (continued)

```
;GRAY CODE to BINARY conversion algorithm
(program gc2s 2
(def 1 ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def reset signal input 5)
(def inp signal input 6 )
(def bin signal output 7)
(def 8  power)
(process grycod  Ø
msbs
    (cond((not inp)(setq bin (not inp))(go msbs))
         (inp      (setq bin inp)(go compl)))
compl
    (cond((not inp)(setq bin  inp)(go compl))
         (inp      (setq bin (not inp))(go nextbit)))
nextbit
    (cond((not inp)(setq bin(not inp))(go nextbit))
         (inp      (setq bin inp)(go compl)))  ) )
```

THIS ALGORITHM EXHIBITS THE GRAY CODE DECODING SCHEME DONE IN THE CONTROL PATH. THE ONLY DATA PATH ORGANELLES INSTANTIATED ARE THOSE ASSOCIATED WITH THE SEQUENCER. THE WIDTH OF THE SEQURNCER (2 BITS) IS DEFINED EXPLICITLY IN THE PROGRAM STATEMENT, EVEN THOUGH NO ACTUAL DATA PATH (AS SUCH) EXISTS. THE IMPLICATION IS THAT FSMs CAN BE CREATED WITHOUT AN "ACTUAL DATA PATH".

Gcs.mac



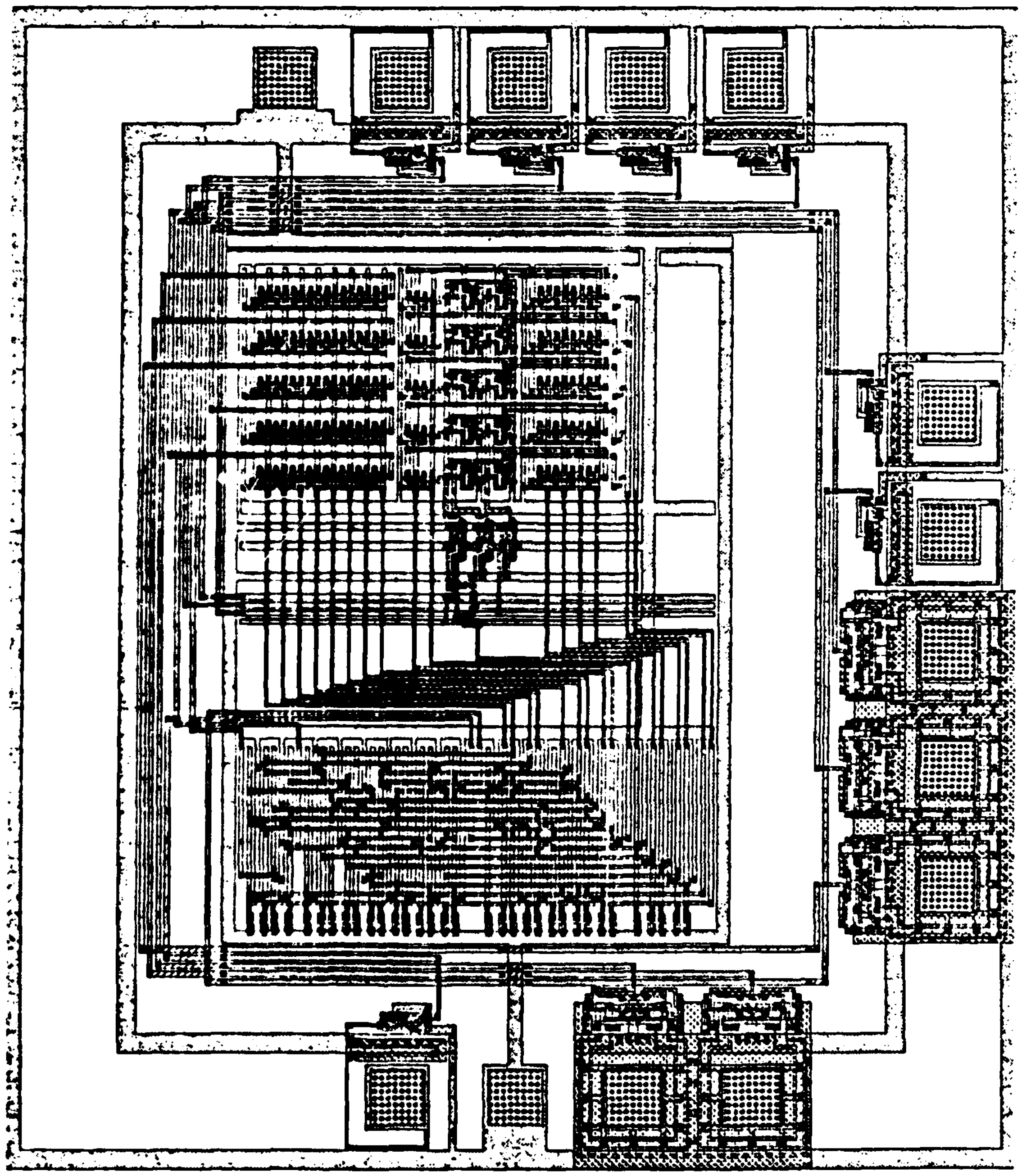
Gcs.cif

```
Statistic - for project gcs
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 65, 55 - Reading source file - gcs.mac
Herald - 70, 55 - Reading library from - /vlsi/macpit/library
Herald - 887, 598 - Processing definitions
Herald - 889, 598 - Evaluating evals
Herald - 975, 598 - Expanding macros
Herald - 995, 598 - Extracting sources
Herald - 1094, 692 - Extracting destinations
Herald - 1095, 692 - Extracting labels
Herald - 1095, 692 - Extracting sequencers
Herald - 1098, 692 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 4
Statistic - Number of gates is 25
Statistic - Data-path has 4 Units
Herald - 2138, 1378 - Outputing .obj file
Herald - 2214, 1378 - Extruding gates
Statistic - Control has 29 columns
Herald - 8365, 4632 - Extruding straps
Statistic - Circuit has 215 transistors
Statistic - Control has 13 tracks
Statistic - Power consumption is 0.041979 Watts
Herald - 8769, 4850 - Laying out data-path
Herald - 8803, 4850 - Organelle unit# 1 bit 1
Herald - 9319, 5181 - Organelle unit# 1 bit 0
Herald - 9397, 5181 - Organelle unit# 2 bit 1
Herald - 9564, 5296 - Organelle unit# 2 bit 0
Herald - 9635, 5296 - Organelle unit# 3 bit 1
Herald - 9891, 5407 - Organelle unit# 3 bit 0
Herald - 10083, 5518 - Organelle unit# 4 bit 1
Herald - 10101, 5518 - Organelle unit# 4 bit 0
Statistic - Data-path internal bus uses 3 tracks
Herald - 10155, 5518 - Laying out control
Herald - 11868, 6353 - Laying out flags
Herald - 11871, 6353 - Laying out river
Herald - 12011, 6469 - Laying out wing
Herald - 12024, 6469 - Laying out skeleton
Herald - 12093, 6469 - Laying out pins
Statistic - Dimensions are 1.742500 mm by 1.942500 mm
Herald - 14192, 7428 - Outputing .cif file
Statistic - Memory used - 377K
Statistic - Compilation took 4.008611 CPU minutes
Statistic - Garbage collection took 2.098333 CPU minutes
Statistic - For a total of 71 garbage collections
```

Gcs.scr

```
;DPLC2.MAC
(program dplc2   5          ;there are 5 outputs
(def 13 power)
(def 1  ground)
(def 2 phia)
(def 3 phib)
(def 4 phic)
(def c signal input 5)    ;note use of Boolean inputs
(def tl signal input 6)
(def ts signal input 7)
(def reset signal input 14)

(def lc port output ( 8 9 1Ø 11 12)) ;and integer outputs

(process light_controller  Ø ;stipulates FSM architecture
hg                           ;HIGHWAY GREEN state
        (cond((not(and c tl ) )  ;if TRUE,set these outputs.
                               (setq lc 4)
                                           (go hg))

              (t               (setq lc 5)
                                           (go hy)) )

hy                           ;HIGHWAY YELLOW state
        (cond((not ts)
                               (setq lc 12)
                                           (go hy))

              (t               (setq lc 13)
                                           (go fg))   )

fg                           ;FARMROAD GREEN state
       (cond((not(or tl(not c)))
                               (setq lc 16)
                                           (go fg))

             (t                (setq lc 17 )
                                           (go fy))   )

fy                           ;FARMROAD YELLOW state
      (cond((not ts)
                               (setq lc 18)
                                           (go fy))

            (t                 (setq lc 19)
                                           (go hg))   ) ))))
```

Dplc2.mac



Dplc2.cif

```
Statistic - for project dplc2
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 62, 55 - Reading source file - dplc2.mac
Herald - 68, 55 - Reading library from - /vlsi/macpit/library
Herald - 905, 604 - Processing definitions
Herald - 906, 604 - Evaluating evals
Herald - 989, 604 - Expanding macros
Herald - 1107, 702 - Extracting sources
Herald - 1111, 702 - Extracting destinations
Herald - 1114, 702 - Extracting labels
Herald - 1114, 702 - Extracting sequencers
Herald - 1117, 702 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 5
Statistic - Number of gates is 34
Statistic - Data-path has 4 Units
Herald - 2277, 1498 - Outputing .obj file
Herald - 2410, 1498 - Extruding gates
Statistic - Control has 40 columns
Herald - 8931, 4725 - Extruding straps
Statistic - Circuit has 346 transistors
Statistic - Control has 17 tracks
Statistic - Power consumption is 0.056716 Watts
Herald - 9580, 5048 - Laying out data-path
Herald - 9922, 5267 - Organelle unit# 1 bit 4
Herald - 10156, 5379 - Organelle unit# 1 bit 3
Herald - 10207, 5379 - Organelle unit# 1 bit 2
Herald - 10375, 5498 - Organelle unit# 1 bit 1
Herald - 10533, 5607 - Organelle unit# 1 bit 0
Herald - 10859, 5718 - Organelle unit# 2 bit 4
Herald - 11242, 5928 - Organelle unit# 2 bit 3
Herald - 11266, 5928 - Organelle unit# 2 bit 2
Herald - 11291, 5928 - Organelle unit# 2 bit 1
Herald - 11316, 5928 - Organelle unit# 2 bit 0
Herald - 11552, 6042 - Organelle unit# 3 bit 4
Herald - 11590, 6042 - Organelle unit# 3 bit 3
Herald - 11722, 6148 - Organelle unit# 3 bit 2
Herald - 11748, 6148 - Organelle unit# 3 bit 1
Herald - 11777, 6148 - Organelle unit# 3 bit 0
Herald - 12052, 6272 - Organelle unit# 4 bit 4
Herald - 12068, 6272 - Organelle unit# 4 bit 3
Herald - 12080, 6272 - Organelle unit# 4 bit 2
Herald - 12204, 6383 - Organelle unit# 4 bit 1
Herald - 12216, 6383 - Organelle unit# 4 bit 0
Statistic - Data-path internal bus uses 2 tracks
Herald - 12313, 6383 - Laying out control
Herald - 14457, 7438 - Laying out flags
Herald - 14461, 7438 - Laying out river
Herald - 14506, 7438 - Laying out wing
Herald - 14521, 7438 - Laying out skeleton
Herald - 14578, 7438 - Laying out pins
Statistic - Dimensions are 2.160000 mm by 2.460000 mm
Herald - 18275, 9184 - Outputing .cif file
Statistic - Memory used - 414K
Statistic - Compilation took 5.164444 CPU minutes
Statistic - Garbage collection took 2.586667 CPU minutes
Statistic - For a total of 86 garbage collections
```

Dplc2.scr

APPENDIX C

CHAPTER V LISTINGS

```
Script started on Sat Jun 15 15:14:27 1985
% /vlsi/berk85/bin/crystal splacis.sim
Crystal, v.2
: build splacis.sim
[0:00.5u 0:00.2s 31k]
: inpits c tl ts phia phib
Unknown command: inpits
: inputs c tl ts phia phib
[0:00.0u 0:00.1s 40k]
: outputs st hl0 hl1 fl0 fl1
[0:00.0u 0:00.0s 40k]
: delay phia 0 -1
Marking transistor flow...
Setting Vdd to 1...
Setting GND to 0...
(198 stages examined.)
[0:00.5u 0:00.1s 47k]
: critical
Node hl0 is driven high at 26.93ns
      ...through fet at (154, -155) to Vdd after
    50 is driven low at 23.99ns
      ...through fet at (158, -106) to 93
      ...through fet at (156, -59) to GND after
    156 is driven high at 18.05ns
      ...through fet at (5, -61) to Vdd after
    73 is driven low at 9.33ns
      ...through fet at (69, -113) to GND after
    41 is driven high at 6.31ns
      ...through fet at (75, -124) to Vdd after
    27 is driven low at 1.95ns
      ...through fet at (76, -153) to 4
      ...through fet at (119, -126) to GND after
    phia is driven high at 0.00ns
[0:00.1u 0:00.0s 47k]
: critical -g splaphia
[0:00.0u 0:00.1s 52k]
: clear
[0:00.0u 0:00.0s 52k]
: delay phib 0 -1
Marking transistor flow...
Setting Vdd to 1...
Setting GND to 0...
(126 stages examined.)
[0:00.3u 0:00.0s 52k]
: critical
Node hl0 is driven high at 32.06ns
      ...through fet at (154, -155) to Vdd after
    50 is driven low at 29.11ns
      ...through fet at (158, -106) to 93
      ...through fet at (156, -59) to GND after
    156 is driven high at 23.17ns
      ...through fet at (5, -61) to Vdd after
    73 is driven low at 14.46ns
      ...through fet at (69, -113) to GND after
    41 is driven high at 11.43ns
      ...through fet at (75, -124) to Vdd after
    27 is driven low at 6.97ns
      ...through fet at (76, -153) to 4
```

Crystal TIming Analysis of -Cis PLA

```
      ...through fet at (119, -126) to GND after
    59 is driven high at 2.67ns
      ...through fet at (118, -1Ø6) to 88
      ...through fet at (117, 11) to Vdd after
    phib is driven high at Ø.ØØns
[Ø:ØØ.1u Ø:ØØ.1s 52k]
: critical -g splaphib
[Ø:ØØ.1u Ø:ØØ.1s 52k]
: quit
[Ø:Ø1.7u Ø:ØØ.5s 52k] Crystal done.
% ^D
script done on Sat Jun 15 15:16:58 1985
```

Crystal Analysis of -Cis PLA (continued)

```
Script started on Sat Jun 15 15:18:00 1985
% /vlsi/berk85/bin/crystal lt.sim
Crystal, v.2
: build lt.sim
[0:00.8u 0:00.2s 39k]
: inputs phia phib c tl ts
[0:00.0u 0:00.1s 48k]
: outputs st fl0 fl1 hl0 hl1
[0:00.0u 0:00.0s 48k]
: delay phia 0 -1
Marking transistor flow...
Setting Vdd to 1...
Setting GND to 0...
(21 stages examined.)
[0:00.7u 0:00.1s 50k]
: critical
Node 228 is driven low at 10.16ns
      ...through fet at (569, 453) to 262
      ...through fet at (568, 570) to 88
      ...through fet at (456, 538) to 411
      ...through fet at (480, 537) to GND after
    260 is driven high at 4.92ns
      ...through fet at (416, 930) to Vdd after
    533 is driven low at 0.75ns
      ...through fet at (365, 942) to GND after
    phia is driven high at 0.00ns
[0:00.1u 0:00.0s 50k]
: critical -g ltphia
[0:00.0u 0:00.1s 55k]
: clear
[0:00.0u 0:00.0s 55k]
: delay phib 0 -1
(221 stages examined.)
[0:00.8u 0:00.0s 60k]
: critical
Node st is driven low at 135.82ns
      ...through fet at (911, 583) to GND after
    373 is driven high at 133.89ns
      ...through fet at (893, 510) to Vdd after
    398 is driven low at 131.02ns
      ...through fet at (866, 570) to GND after
    364 is driven high at 123.52ns
      ...through fet at (877, 510) to Vdd after
    76 is driven low at 108.50ns
      ...through fet at (584, 411) to 88
      ...through fet at (478, 435) to 201
      ...through fet at (472, 415) to GND after
    190 is driven high at 15.03ns
      ...through fet at (479, 406) to 163
      ...through fet at (666, 930) to Vdd after
    181 is driven high at 4.91ns
      ...through fet at (541, 930) to Vdd after
    535 is driven low at 0.75ns
      ...through fet at (490, 942) to GND after
    phib is driven high at 0.00ns
[0:00.1u 0:00.1s 60k]
: critical -g ltphib
[0:00.1u 0:00.0s 65k]
```

Crystal Analysis of FLA Light Controller Chip

```
Script started on Thu Jun 13 23:30:02 1985
% powest -p < lt.sim
gamma=0.4V**.5, tox=9e-08m, u0=0.08m**2/V-s
vdd=5V, vtd=-3.5V, vte=0.8V, vsb=2V
#devs    Pdc_avg (W)       Pdc_max (W)       type

0        0.000000          0.000000          enhancement pullups
20       0.011980          0.023959          depletion pullups
15       0.030536          0.061072          special depletion pullups

35       0.042516          0.085032          TOTAL
% ^D
script done on Thu Jun 13 23:31:12 1985
```

Powest Analysis of PLA Light Controller Chip

```
%/vlsi/berk85/bin/crystal stop.sim
:inputs c tl ts rst
:outputs st hlØ hl1 flØ fl1
:set 1 phia phic
:delay phib Ø -1
:critical            (9.6ns)
:clear
:set 1 phia
:delay phib -1 Ø
:delay phic -1 Ø
:critical            (56.67ns)
:clear
:set Ø phib phic
:delay phia -1 Ø
:critical            (17.55ns)
:clear
:set Ø phib phic
:delay phia Ø -1
:critical            (54.63ns)
:clear
:set 1 phia
:set Ø phib
:delay phic Ø -1
:critical            (363.52ns)
:quit
```

Crystal Command File for MacPitts Light Controller Chip

APPENDIX D

CHAPTER VI LISTINGS

```
Statistic - for project ham15.4
Statistic - options: (herald opt-d opt-c stat obj cif nologo)
Herald - 59, 52 - Reading source file - ham15.4.mac
Herald - 78, 52 - Reading library from - /vlsi/macpit/library
Herald - 890, 591 - Processing definitions
Herald - 894, 591 - Evaluating evals
Herald - 980, 591 - Expanding macros
Herald - 2822, 1405 - Extracting sources
Herald - 2982, 1511 - Extracting destinations
Herald - 3015, 1511 - Extracting labels
Herald - 3015, 1627 - Extracting sequencers
Herald - 3131, 1627 - Extracting flags, data-path, control, and pins
Statistic - Maximum control depth is 7
Statistic - Number of gates is 140
Statistic - Data-path has 0 Units
Herald - 9964, 4968 - Outputing .obj file
Herald - 10373, 4968 - Extruding gates
Statistic - Control has 155 columns
Herald - 586415, 233036 - Extruding straps
Statistic - Circuit has 715 transistors
Statistic - Control has 42 tracks
Statistic - Power consumption is 0.160860 Watts
Herald - 589965, 234452 - Laying out data-path
Statistic - Data-path internal bus uses 0 tracks
Herald - 589967, 234452 - Laying out control
Herald - 599196, 239812 - Laying out flags
Herald - 599197, 239812 - Laying out river
Herald - 599206, 239812 - Laying out wing
Herald - 599281, 239812 - Laying out skeleton
Herald - 599325, 239812 - Laying out pins
Statistic - Dimensions are 5.137500 mm by 4.005000 mm
Herald - 606259, 242522 - Outputing .cif file
Statistic - Memory used - 529K
Statistic - Compilation took 168.593902 CPU minutes
Statistic - Garbage collection took 67.456947 CPU minutes
Statistic - For a total of 1805 garbage collections
```

Ham15dc.scr

LIST OF REFERENCES

1. Conradi, J. R. and Hauenstein, B. R., VLSI Design of a Very Fast Pipeline Carry Look Ahead Adder, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1983.

2. Carlson, D. J., Application of a Silicon Compiler to VLSI Design of Digital Pipelined Multipliers, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1984.

3. Froede, A. O., Silicon Compiler Design of Combinational and Pipeline Adder Integrated Circuits, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1985.

4. Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, 1980.

5. Hamming, R., Coding and Information Theory, Prentice-Hall, 1980.

6. Lincoln Laboratory, Massachusetts Institute of Technology Project Report RVLSI-3, An Introduction to MacPitts, by J. R. Southard, 10 February 1983.

8. Weinberger, A., "Large Scale Integration of MOS Complex Logic: A Layout Method", IEEE Journal of Solid State Circuits, v. sc-2, pp. 182-190 December 1967.

9. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, EQNTOTT, by J. Ousterhout, pp. 1-6, 1981 .

10. Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, TPLA, by R. N. Mayo, pp. 1-5, 1983.

11. Newkirk, J., and Mathews, R., The VLSI Designer's Library, Addison-Wesley, 1983.

12. University of California Regents, The Franz LISP Manual, by J. K. Foderaro, p.11-1, 1980.

13. McEliece, R. J., Encyclopedia of Mathematics and Its Applications, vol. 3, (Probability, The Theory of Information and Coding), pp. 142-146, Addison-Wesley, 1977.

14. Rabiner, L.R., and Gold, B., Theory and Application of Digital Signal Processing, pp.541, Prentice-Hall, 1975.

BIBLIOGRAPHY

Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memo. No. ERL-M520, SPICE2: A Computer Program to Simulate Semiconductor Circuits, by L. Nagel, 9 May 1975.

Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memo. No. ERL-M592, Program Reference for SPICE2, by Ellis Cohen, 14 June 1976.

Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Memo. No. UCB/ERL M80/7, The Simulation of MOS Integrated Circuits Using SPICE2, February 1980 (Revised October 1980).

Mavor, J. , Denyer, P. B., and Jack, M. A., INTRODUCTION to MOS LSI DESIGN, Addison-Wesley, 1983.

Muroga, S., VLSI System Design, Wiley-Interscience, 1982.

Ullman, J. D., Computational Aspects of VLSI, Computer Science Press, 1984.

Waite, M., Martin, D., and Prata, S., UNIX Primer Plus, H. Sams Co., 1983.

Winston, P. H., and Horn, B. K., LISP, Addison-Wesley, 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
2.	Dr. Donald Kirk Code 62KI Naval Postgraduate School Monterey, California 93943-5100	5
3.	Dr. H. R. Loomis Code 62LM Naval Postgraduate School Monterey, California 93943-5100	2
4.	Dr. J. Fredericksen Code 63FS Naval Postgraduate School Monterey, California 93943-5100	1
5.	Chairman, ECE Department, Code 62RR Naval Postgraduate School Monterey, California 93943-5100	2
6.	Mr. Antun Domic, B-347 Massachusetts Institute of Technology Lincoln Laboratory P.O. Box 73 Lexington, Massachusetts 02173-0073	1
7.	Mr. J. R. Southard MetaLogic, Inc. 725 Concord Ave. Cambridge, Masachusetts 02138	1
8.	Robert C. Larrabee 5313 Angus Dr. Virginia Beach, Virginia 23464	1
9.	Mr. R. N. Larrabee P.O. Box 96 Hollywood, Maryland 20636	1
10.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2

11.	Dr. Robert Kenedy Code 52KC Naval Postgraduate School Monterey, California 93943-5100	1
12.	CPT Brad Mercer Code 52ZI Naval Postgraduate School Monterey, California 93943-5100	1